Contents

- Documentación de C#
- Primeros pasos
  - Información general
  - Introducción al lenguaje C# y .NET Framework
- Tutoriales
  - Información general
  - Introducción a la programación con C#
    - Elección de la primera lección
    - Hola a todos
    - Números en C#
    - Bifurcaciones y bucles
    - Colecciones de listas
    - Trabajo en un entorno local
      - Configuración del entorno
      - Números en C#
      - Bifurcaciones y bucles
      - Colecciones de listas
      - Introducción a las clases
  - Exploración de C# 6
  - Exploración de la interpolación de cadenas: tutorial interactivo
  - Exploración de la interpolación de cadenas: tutorial en el entorno
  - Escenarios avanzados de la interpolación de cadenas
  - Actualización segura de la interfaz con métodos de la predeterminada
  - Creación de la funcionalidad Mixin con métodos de interfaz predeterminados
  - Exploración de los índices y rangos
  - Trabajo con tipos de referencias que aceptan valores NULL
  - Actualización de aplicaciones a tipos de referencias que aceptan valores NULL
  - Generación y consumo de flujos asincrónicos
  - Aplicación de capacidades de datos con la coincidencia de patrones

- Aplicación de consola
- Cliente REST
- Herencia en C# y .NET
- Uso de LINQ
- Uso de atributos

Paseo por C#

- Introducción
- Estructura del programa
- Tipos y variables
- Expresiones
- Instrucciones
- Clases y objetos
- Matrices
- Interfaces
- Delegados
- Atributos

Novedades de C#

- C# 8.0
- C# 7.3
- C# 7.2
- C# 7.1
- C# 7.0
- C# 6
- Cambios importantes en el compilador
- Historial de versiones de C#
- Relaciones entre el lenguaje y el marco
- Consideraciones sobre versiones y actualizaciones

Conceptos de C#

- Sistema de tipos de C#
- Tipos de referencia que aceptan valores NULL
- Elección de una estrategia para habilitar tipos de referencia que acepten valores NULL
- Espacios de nombres

Tipos básicos
Clases
Tuplas
Deconstruir tuplas y otros tipos
Interfaces
Métodos
Expresiones lambda
Propiedades
Indizadores
Descartes
Genéricos
Iterators
Delegados y eventos
  Introducción a los delegados
  System.Delegate y la palabra clave delegate
  Delegados fuertemente tipados
  Patrones comunes para delegados
  Introducción a los eventos
  Patrón de eventos estándar de .NET
  Patrón de eventos actualizado de .NET
  Distinción de delegados y eventos
Language-Integrated Query (LINQ)
  Información general de LINQ
  Conceptos básicos de las expresiones de consultas
  LINQ en C#
  Escribir consultas LINQ en C#
  Consultar una colección de objetos
  Devolver una consulta de un método
  Almacenar los resultados de una consulta en memoria
  Agrupar los resultados de consultas
  Crear un grupo anidado
  Realizar una subconsulta en una operación de agrupación

Agrupar resultados por claves contiguas
Especificar dinámicamente filtros con predicado en tiempo de ejecución
Realizar combinaciones internas
Realizar combinaciones agrupadas
Realizar operaciones de combinación externa izquierda
Ordenar los resultados de una cláusula join
Realizar una unión usando claves compuestas
Realizar operaciones de combinación personalizadas
Controlar valores nulos en expresiones de consulta
Controlar excepciones en expresiones de consulta
Programación asincrónica
Coincidencia de modelos
Escritura de código seguro y eficaz
Árboles de expresión
Introducción a los árboles de expresión
Árboles de expresiones en detalle
Tipos de marco que admiten árboles de expresión
Ejecución de expresiones
Interpretación de expresiones
Generación de expresiones
Traducción de expresiones
Resumen
Interoperabilidad nativa
Documentación del código
Control de versiones
Artículos de procedimientos de C#
Índice de artículos
Análisis de cadenas mediante `String.Split`
Concatenación de cadenas
Conversión de una cadena en un valor DateTime
Búsqueda en las cadenas
Modificación del contenido de las cadenas

Comparación de cadenas
Conversión segura mediante coincidencia de patrones, operadores is y as
SDK de .NET Compiler Platform (API de Roslyn)
Información general del SDK de .NET Compiler Platform (API de Roslyn)
Descripción del modelo de API de compilador
Trabajar con sintaxis
Trabajar con semántica
Trabajar con un área de trabajo
Exploración de código con el visualizador de sintaxis
Guías rápidas
Análisis de sintaxis
Análisis semántico
Transformación de sintaxis
Tutoriales
Crear el primer analizador y la corrección de código
Guía de programación de C#
Información general
Dentro de un programa de C#
Qué hay dentro de un programa de C#
Hello World, su primer programa
Estructura general de un programa de C#
Nombres de identificador
Convenciones de código de C#
Main() y argumentos de línea de comandos
Información general
Argumentos de la línea de comandos
Procedimiento para mostrar argumentos de la línea de comandos
Valores devueltos Main()
Conceptos de programación
Información general
Programación asincrónica con async y await
Información general

- Modelo de programación asincrónica de tareas
- Tutorial: Acceso a web mediante async y await
- Ampliación del tutorial de async usando Task.WhenAll
- Realización de varias solicitudes web en paralelo utilizando async y await
- Tipos de valor devueltos de Async
- Flujo de control en programas Async
- Cancelación de tareas y procesamiento de tareas completadas
  - Información general
  - Cancelar una tarea asincrónica o una lista de tareas
  - Cancelar tareas asincrónicas tras un período de tiempo
  - Cancelar las tareas asincrónicas restantes cuando se completa una
  - Iniciar varias tareas asincrónicas y procesarlas a medida que se completan
- Controlar la reentrada en aplicaciones asincrónicas
- Usar Async en acceso a archivos

Atributos
- Información general
- Crear atributos personalizados
- Acceso a atributos mediante reflexión
- Procedimiento para: Crear una unión de C/C++ mediante atributos

Colecciones

Covarianza y contravarianza
- Información general
- Varianza en interfaces genéricas
  - Creación de interfaces genéricas variantes
  - Uso de la varianza en interfaces para las colecciones genéricas
- Varianza en delegados
  - Uso de varianza en delegados
  - Uso de varianza para los delegados genéricos Func y Action

Árboles de expresión
- Información general
- Ejecución de árboles de expresión
- Modificación de árboles de expresión

- Uso de árboles de expresión para crear consultas dinámicas
- Depuración de árboles de expresión en Visual Studio
- Sintaxis de DebugView
- Iterators
- Language-Integrated Query (LINQ)
  - Información general
  - Introducción a LINQ en C#
    - Introducción a las consultas LINQ
    - LINQ y tipos genéricos
    - Operaciones básicas de consulta LINQ
    - Transformaciones de datos con LINQ
    - Relaciones entre tipos en las operaciones de consulta LINQ
    - Sintaxis de consulta y sintaxis de método en LINQ
    - Características de C# compatibles con LINQ
    - Tutorial: Escribir consultas en C# (LINQ)
  - Información general sobre operadores de consulta estándar
    - Información general
    - Sintaxis de las expresiones de consulta para operadores de consulta estándar
    - Clasificación de operadores de consulta estándar por modo de ejecución
    - Ordenación de datos
    - Operaciones Set
    - Filtrado de datos
    - Operaciones cuantificadoras
    - Operaciones de proyección
    - Realización de particiones de datos
    - Operaciones de combinación
    - Agrupar datos
    - Operaciones de generación
    - Operaciones de igualdad
    - Operaciones de elementos
    - Conversión de tipos de datos
    - Operaciones de concatenación

Operaciones de agregación
LINQ to Objects
Información general
LINQ y cadenas
Artículos de procedimientos
Procedimiento para: Realizar un recuento de las repeticiones de una palabra en una cadena (LINQ)
Procedimiento para buscar frases que contengan un conjunto especificado de palabras (LINQ)
Procedimiento para consultar caracteres en una cadena (LINQ)
Procedimiento para: Combinar consultas LINQ con expresiones regulares
Procedimiento para buscar la diferencia de conjuntos entre dos listas (LINQ)
Procedimiento para ordenar o filtrar datos de texto por palabra o campo (LINQ)
Procedimiento para reordenar los campos de un archivo delimitado (LINQ)
Procedimiento para: Combinar y comparar colecciones de cadenas (LINQ)
Procedimiento para rellenar colecciones de objetos de varios orígenes (LINQ)
Procedimiento para dividir un archivo en varios mediante el uso de grupos (LINQ)
Procedimiento para combinar contenido de archivos no similares (LINQ)
Procedimiento para: Calcular valores de columna en un archivo de texto CSV (LINQ)
LINQ y Reflection
Procedimiento para consultar los metadatos de un ensamblado con la función de reflexión (LINQ)
LINQ y directorios de archivos
Información general
Procedimiento para buscar archivos con un nombre o atributo especificados
Procedimiento para agrupar archivos por extensión (LINQ)
Procedimiento para consultar el número total de bytes en un conjunto de carpetas (LINQ)
Procedimiento para: Comparar el contenido de dos carpetas (LINQ)
Procedimiento para consultar el archivo o archivos de mayor tamaño en un árbol de directorios (LINQ)

Procedimiento para consultar archivos duplicados en un árbol de directorios (LINQ)
Cómo: Consultar el contenido de los archivos de una carpeta (LINQ)
Procedimiento para consultar un objeto ArrayList con LINQ
Procedimiento para: Agregar métodos personalizados para las consultas LINQ
LINQ to XML
Introducción (LINQ to XML)
Información general de LINQ to XML
LINQ to XML frente a DOM
LINQ to XML frente a otras tecnologías XML
Guía de programación (LINQ to XML)
Información general sobre la programación de LINQ to XML
Árboles XML
Trabajar con espacios de nombres XML
Serializar árboles XML
Ejes de LINQ to XML
Consultar árboles XML
Modificar árboles XML (LINQ to XML)
Rendimiento (LINQ to XML)
Programación avanzada de LINQ to XML
Seguridad de LINQ to XML
Documentos XML de ejemplo (LINQ to XML)
Referencia (LINQ to XML)
LINQ to ADO.NET (Página de portal)
Habilitar un origen de datos para realizar consultas LINQ
Características de IDE de Visual Studio y herramientas para LINQ
Programación orientada a objetos
Reflexión
Serialización (C#)
Información general
Procedimiento para escribir objetos de datos en un archivo XML
Procedimiento para leer objetos de datos de un archivo XML
Tutorial: Conservar un objeto en Visual Studio

Instrucciones, expresiones y operadores
- Información general
- Instrucciones
- Expresiones
- Miembros con forma de expresión
- Funciones anónimas
  - Información general
  - Expresiones lambda
  - Procedimiento para usar expresiones lambda en una consulta
- Igualdad y comparaciones de igualdad
  - Comparaciones de igualdad
  - Procedimiento para definir la igualdad de valores para un tipo
  - Procedimiento para probar la igualdad de referencia (Identidad)

Tipos
- Uso y definición de tipos
- Conversiones de tipos
- Conversión boxing y conversión unboxing
- Procedimiento para convertir una matriz de bytes en un valor int
- Procedimiento para convertir una cadena en un número
- Procedimiento para convertir cadenas hexadecimales en tipos numéricos
- Uso de tipo dinámico
- Tutorial: Crear y usar objetos dinámicos (C# y Visual Basic)

Clases y structs
- Información general
- Clases
- de la empresa
- Herencia
- Polimorfismo
  - Información general
  - Control de versiones con las palabras clave Override y New
  - Saber cuándo utilizar las palabras clave Override y New
  - Invalidación del método ToString

Miembros
- Información general sobre los miembros
- Clases y miembros de clase abstractos y sellados
- Clases estáticas y sus miembros
- Modificadores de acceso
- Campos
- Constantes
- Definición de propiedades abstractas
- Definición de constantes en C#

Propiedades
- Información general sobre propiedades
- Utilizar propiedades
- Propiedades de interfaz
- Restringir la accesibilidad del descriptor de acceso
- Declaración y uso de propiedades de lectura y escritura
- Propiedades autoimplementadas
- Cómo implementar una clase ligera con propiedades autoimplementadas

Métodos
- Información general sobre los métodos
- Funciones locales
- Valores devueltos y variables locales de tipo ref
- Parámetros
  - Pasar parámetros
  - Pasar parámetros de tipo de valor
  - Pasar parámetros Reference-Type
  - Conocimiento de las diferencias entre pasar a un método un struct y una referencia a clase
- Variables locales con asignación implícita de tipos
- Uso de matrices y variables locales con tipo implícito en expresiones de consulta
- Métodos de extensión.
- Implementación e invocación de un método de extensión personalizado
- Creación de un método nuevo para una enumeración
- Argumentos opcionales y con nombre

Uso de argumentos opcionales y con nombre en la programación de Office
Constructores
Información general sobre los constructores
Utilizar constructores
Constructores de instancias
Constructores privados
Constructores estáticos
Escritura de un constructor copy
Finalizadores
Inicializadores de objeto y colección
Procedimiento para inicializar un objeto mediante un inicializador de objeto
Procedimiento para inicializar un diccionario con un inicializador de colección
Tipos anidados
Clases y métodos parciales
Tipos anónimos
Devolución de subconjuntos de propiedades de elementos en una consulta
Interfaces
Información general
Implementación explícita de interfaz
Procedimiento para implementar miembros de interfaz de forma explícita
Procedimiento para implementar miembros de dos interfaces de forma explícita
Delegados
Información general
Utilizar delegados
Delegados con métodos con nombre y Métodos anónimos
Procedimiento para combinar delegados (delegados de multidifusión) (Guía de programación de C#)
Procedimiento para declarar un delegado, crear instancias del mismo y usarlo
Matrices
Información general
Matrices como objetos
Matrices unidimensionales
Matrices multidimensionales

Matrices escalonadas
Usar foreach con matrices
Pasar matrices como argumentos
Matrices con tipo implícito
Cadenas
Programación con cadenas
Procedimiento para determinar si una cadena representa un valor numérico
Indizadores
Información general
Utilizar indizadores
Indizadores en interfaces
Comparación entre propiedades e indizadores
Events
Información general
Procedimiento para suscribir y cancelar la suscripción a eventos
Procedimiento para publicar eventos que cumplan las instrucciones de .NET Framework
Procedimiento para generar eventos de una clase base en clases derivadas
Procedimiento para implementar eventos de interfaz
Procedimiento para implementar descriptores de acceso de eventos personalizados
Genéricos
Información general
Parámetros de tipos genéricos
Restricciones de tipos de parámetros
Clases genéricas
Interfaces genéricas
Métodos genéricos
Genéricos y matrices
Delegados genéricos
Diferencias entre plantillas de C++ y tipos genéricos de C#
Genéricos en el tiempo de ejecución
Genéricos y reflexión
Genéricos y atributos

Espacios de nombres
Información general
Utilizar espacios de nombres
Procedimiento para usar el espacio de nombres My
Código no seguro y punteros
Información general y restricciones
Búferes de tamaño fijo
Tipos de puntero
Información general
Conversiones de puntero
Procedimiento para usar punteros para copiar una matriz de bytes
Comentarios de la documentación XML
Información general
Etiquetas recomendadas para los comentarios de la documentación
Procesamiento del archivo XML
Delimitadores para etiquetas de documentación
Procedimiento para usar las características de la documentación XML
Referencia de la etiqueta de documentación
<c>
<code>
Atributo cref
<example>
<exception>
<include>
<inheritdoc>
<list>
<para>
<param>
<paramref>
<permission>
<remarks>
<returns>

<see>
<seealso>
<summary>
<typeparam>
<typeparamref>
<value>
Excepciones y control de excepciones
Información general
Utilizar excepciones
Control de excepciones
Creación y producción de excepciones
Excepciones generadas por el compilador
Procedimiento para controlar una excepción mediante Try y Catch
Procedimiento para ejecutar código de limpieza mediante finally
Procedimiento para detectar excepciones no compatibles con CLS
Registro y sistema de archivos
Información general
Procedimiento para recorrer en iteración un árbol de directorio
Procedimiento para obtener información sobre archivos, carpetas y unidades
Procedimiento para crear archivos o carpetas
Procedimiento para copiar, eliminar y mover archivos y carpetas
Procedimiento para proporcionar un cuadro de diálogo de progreso para operaciones de archivos
Procedimiento para escribir en un archivo texto
Procedimiento para leer de un archivo de texto
Procedimiento para leer un archivo de texto línea a línea
Procedimiento para crear una clave en el Registro
Interoperabilidad
Interoperabilidad de .NET
Información general sobre interoperabilidad
Procedimiento para acceder a objetos de interoperabilidad de Office mediante características de C#
Procedimiento para usar propiedades indizadas en la programación de

interoperabilidad COM
Procedimiento para usar la invocación de plataforma para reproducir un archivo WAV
Tutorial: Programación de Office (C# y Visual Basic)
Clase COM de ejemplo
Referencia del lenguaje
Información general
Configurar la versión del lenguaje
Palabras clave de C#
Información general
Tipos
Tipos de valor
Información general
Tipos numéricos integrales
Tipos numéricos de punto flotante
Conversiones numéricas integradas
bool
char
Tipos de enumeración
Tipos de estructura
Tipos de valor que aceptan valores NULL
Tipos de referencia
Características de los tipos de referencia
Tipos de referencia integrados
clase
interfaz
Tipos de referencia que aceptan valores NULL
void
var
Tipos integrados
Tipos no administrados
Valores predeterminados
Modificadores

Modificadores de acceso
Referencia rápida
Niveles de accesibilidad
Dominio de accesibilidad
Restricciones en el uso de los niveles de accesibilidad
internal
private
protected
public
protected internal
privado protegido
abstract
async
const
event
extern
in (modificador genérico)
new (modificador de miembro)
out (modificador genérico)
override
readonly
sealed
static
unsafe
virtual
volatile
Palabras clave de instrucciones
Categorías de instrucciones
Instrucciones de selección
if-else
switch
Instrucciones de iteración
do

- do
- for
- foreach, in
- while
- Instrucciones de salto
  - break
  - continue
  - goto
  - return
- Instrucciones para el control de excepciones
  - throw
  - try-catch
  - try-finally
  - try-catch-finally
- Checked y unchecked
  - Información general
  - checked
  - unchecked
- Instrucción fixed
- Instrucción lock

Parámetros de métodos
- Pasar parámetros
- params
- in (modificador de parámetro)
- ref
- out (modificador de parámetro)

Palabras clave del espacio de nombres
- namespace
- using
  - Contextos de uso
  - using (directiva)
  - Directiva using static
  - Instrucción using

alias externo
Palabras clave de prueba de tipos
is
Palabras clave de restricción de tipo genérico
Restricción new
where
Palabras clave de acceso
base
this
Palabras clave de literales
null
true y false
default
Palabras clave contextuales
Referencia rápida
add
get
partial (Tipos)
partial (método)
remove
set
when (condición de filtro)
value
yield
Palabras clave para consultas
Referencia rápida
from (cláusula)
where (cláusula)
select (cláusula)
group (cláusula)
into
orderby (cláusula)

join (cláusula)
let (cláusula)
ascending
descending
on
equals
by
in
Operadores y expresiones de C#
Información general
Operadores aritméticos
Operadores lógicos booleanos
Operadores de desplazamiento y bit a bit
Operadores de igualdad
Operadores de comparación
Operadores y expresiones de acceso a miembros
Operadores de prueba de tipos y expresión de conversión
Operadores de conversión definidos por el usuario
Operadores relacionados con punteros
Operadores de asignación
+ Operadores and +=
- Operadores and -=
Operador ?:
! (permite valores NULL)
?? Operadores and ??=
Operador =>
:: (operador)
await (operador)
Expresiones de valor predeterminado
operador delegate
Expresión nameof
Operador new

Operador sizeof
Expresión stackalloc
Expresión switch
operadores true y false
Sobrecarga de operadores
Caracteres especiales de C#
Información general
$ -- interpolación de cadenas
@ -- identificador textual
Atributos leídos por el compilador
Atributos globales
General
Información del agente de llamada
Análisis estático que admite un valor NULL
Directivas de preprocesador de C#
Información general
#if
#else
#elif
#endif
#define
#undef
#warning
#error
#line
#region
#endregion
#pragma
#pragma warning
#pragma checksum
Opciones del compilador de C#
Información general

Compilar la línea de comandos con csc.exe
Establecimiento de variables de entorno para la línea de comandos de Visual Studio
Opciones del compilador de C#, por categoría
Opciones del compilador de C#, por orden alfabético
@
-addmodule
-appconfig
-baseaddress
-bugreport
-checked
-codepage
-debug
-define
-delaysign
-deterministic
-doc
-errorreport
-filealign
-fullpaths
-help, -?
-highentropyva
-keycontainer
-keyfile
-langversion
-lib
-link
-linkresource
-main
-moduleassemblyname
-noconfig
-nologo
-nostdlib

- -nowarn
- -nowin32manifest
- -nullable
- -optimize
- -out
- -pathmap
- -pdb
- -platform
- -preferreduilang
- -publicsign
- -recurse
- -reference
- -refout
- -refonly
- -resource
- -subsystemversion
- -target
  - -target:appcontainerexe
  - -target:exe
  - -target:library
  - -target:module
  - -target:winexe
  - -target:winmdobj
- -unsafe
- -utf8output
- -warn
- -warnaserror
- -win32icon
- -win32manifest
- -win32res

Errores del compilador de C#

Borrador de especificación de C# 6.0

Propuestas de C# 7.0 - 8.0

Tutoriales

Introducción a C#

18/03/2020 • 3 minutes to read • Edit Online

Esta sección proporciona breves y sencillos tutoriales simples que permiten compilar una aplicación con rapidez mediante C# y .NET Core. Incluye temas de introducción a Visual Studio y Visual Studio Code. Estos artículos están destinados a usuarios con cierta experiencia en programación. Si no está familiarizado con la programación, consulte nuestros tutoriales interactivos de introducción a C#.

Los siguientes temas se encuentran disponibles:

- Introducción al lenguaje C# y .NET Framework

  Proporciona información general sobre el lenguaje C# y .NET.

- Creación de una aplicación Hola mundo de C# con .NET Core en Visual Studio

  Visual Studio permite codificar, compilar, ejecutar, depurar y publicar aplicaciones para Windows o Mac desde un entorno de desarrollo integrado, así como generar perfiles de dichas aplicaciones.

  El tema le permite crear y ejecutar una aplicación Hola mundo sencilla y, después, modificarla para ejecutar una aplicación Hola mundo algo más interactiva. Una vez que haya terminado de generar y ejecutar la aplicación, también puede aprender cómo depurarla y publicarla para que se pueda ejecutar en cualquier plataforma compatible con .NET Core.

- Creación de una biblioteca de clases con C# y .NET Standard en Visual Studio

  Una biblioteca de clases permite definir los tipos y miembros de tipo que se pueden llamar desde otra aplicación. Este tema le permite crear una biblioteca de clases con un único método que determina si una cadena comienza con un carácter en mayúscula. Una vez que haya terminado de compilar la biblioteca, puede desarrollar una prueba unitaria para asegurarse de que funciona según lo previsto y, a continuación, puede hacer que esté disponible para aplicaciones que desean usarla.

- Introducción a C# y Visual Studio Code

  Visual Studio Code es un editor de código gratuito optimizado para la compilación y depuración de aplicaciones web y en la nube modernas. Es compatible con IntelliSense y está disponible para Windows, macOS y Linux.

  En este tema se muestra cómo crear y ejecutar una aplicación Hola mundo sencilla con Visual Studio Code y .NET Core.

Secciones relacionadas

- Guía de programación de C#

  Proporciona información sobre conceptos de programación en C# y describe cómo realizar diversas tareas en C#.

- Referencia de C#

  Proporciona información de referencia detallada sobre palabras clave, operadores, directivas de preprocesador, opciones del compilador y errores del compilador y advertencias de C#.

- Tutoriales

  Proporciona vínculos a los tutoriales de programación que usan C# y una breve descripción de cada uno.

Vea también

- Desarrollo de C# con Visual Studio

Introducción al lenguaje C# y .NET Framework

16/04/2020 • 10 minutes to read • Edit Online

C# es un lenguaje elegante, con seguridad de tipos y orientado a objetos que permite a los desarrolladores crear una gran variedad de aplicaciones seguras y sólidas que se ejecutan en .NET Framework. Puede usar C# para crear aplicaciones cliente de Windows, servicios web XML, componentes distribuidos, aplicaciones cliente-servidor, aplicaciones de base de datos y muchas, muchas más cosas. Visual C# proporciona un editor de código avanzado, prácticos diseñadores de interfaz de usuario, un depurador integrado y muchas otras herramientas que facilitan el desarrollo de aplicaciones basadas en el lenguaje C# y .NET Framework.

> **NOTE**
>
> En la documentación de Visual C# se supone que comprende los conceptos básicos de programación. Si es principiante, puede que quiera examinar Visual C# Express, que está disponible en Internet. También puede aprovechar los libros y recursos web sobre C# para aprender sobre habilidades prácticas de programación.

lenguaje C#

La sintaxis de C# es muy expresiva, pero también sencilla y fácil de aprender. Cualquier persona familiarizada con C, C++ o Java, reconocerá al instante la sintaxis de llaves de C#. Los desarrolladores que conocen cualquiera de estos lenguajes pueden empezar normalmente a trabajar en C# de forma productiva en un espacio breve de tiempo. La sintaxis de C# simplifica muchas de las complejidades de C++ y proporciona características eficaces, como tipos que aceptan valores NULL, enumeraciones, delegados, expresiones lambda y acceso directo a memoria. C# admite métodos y tipo genéricos, que proporcionan una mayor seguridad de tipos y rendimiento, e iteradores, que permiten a los implementadores de clases de colecciones definir comportamientos de iteración personalizados que son fáciles de usar por el código de cliente. Las expresiones de Language Integrated Query (LINQ) convierten la consulta fuertemente tipada en una construcción de lenguaje de primera clase.

En cuanto lenguaje orientado a objetos, C# admite los conceptos de encapsulación, herencia y polimorfismo. Todas las variables y métodos, incluido el método `Main`, el punto de entrada de la aplicación, se encapsulan dentro de las definiciones de clase. Una clase puede heredar directamente de una clase primaria, pero puede implementar cualquier número de interfaces. Los métodos que invalidan los métodos virtuales en una clase primaria requieren la palabra clave `override` como una manera de evitar redefiniciones accidentales. En C#, un struct es como una clase sencilla; es un tipo asignado en la pila que puede implementar interfaces pero que no admite herencia.

Además de estos principios básicos orientados a objetos, C# facilita el desarrollo de componentes de software mediante varias construcciones de lenguaje innovadoras, incluidas las siguientes:

- Signaturas de método encapsulado llamadas *delegados*, que permiten notificaciones de eventos con seguridad de tipos.
- Propiedades, que actúan como descriptores de acceso para variables miembro privadas.
- Atributos, que proporcionan metadatos declarativos sobre tipos en tiempo de ejecución.
- Comentarios de doc.umentación XML insertados
- Language Integrated Query (LINQ), que proporciona funcionalidades de consulta integradas en diversos orígenes de datos.

Si tiene que interactuar con otro software de Windows, como objetos COM o archivos DLL nativos de Win32, puede hacerlo en C# mediante un proceso denominado "Interoperabilidad". La interoperabilidad permite que los

programas de C# hagan casi todo lo que puede hacer una aplicación C++ nativa. C# admite incluso el uso de punteros y el concepto de código "no seguro" en los casos en los que el acceso directo a memoria es crítico.

El proceso de compilación de C# es simple en comparación con C y C++ y más flexible que en Java. No hay ningún archivo de encabezado independiente y ningún requisito de declaración de métodos y tipos en un orden en particular. Un archivo de código fuente de C# puede definir cualquier número de clases, structs, interfaces y eventos.

Los siguientes son recursos adicionales de C#:

- Para obtener una buena introducción general al lenguaje, consulte el capítulo 1 de la especificación del lenguaje C#.
- Para más información sobre aspectos específicos del lenguaje C#, consulte la referencia de C#.
- Para más información sobre LINQ, consulte LINQ (Language-Integrated Query).

Arquitectura de la plataforma .NET Framework

Los programas de C# se ejecutan en .NET Framework, un componente integral de Windows que incluye un sistema de ejecución virtual denominado Common Language Runtime (CLR) y un conjunto unificado de bibliotecas de clases. El CLR es la implementación comercial de Microsoft de Common Language Infrastructure (CLI), un estándar internacional que es la base para la creación de entornos de ejecución y desarrollo en los que los lenguajes y las bibliotecas trabajan juntos sin problemas.

El código fuente escrito en C# se compila en un lenguaje intermedio (IL) que guarda conformidad con la especificación de CLI. El código y los recursos IL, como mapas de bits y cadenas, se almacenan en disco en un archivo ejecutable denominado ensamblado, normalmente con la extensión .exe o .dll. Un ensamblado contiene un manifiesto que proporciona información sobre los tipos, la versión, la referencia cultural y los requisitos de seguridad del ensamblado.

Cuando se ejecuta el programa de C#, el ensamblado se carga en el CLR, el cual podría realizar diversas acciones en función de la información en el manifiesto. Luego, si se cumplen los requisitos de seguridad, el CLR realiza la compilación Just-In-Time (JIT) para convertir el código IL en instrucciones máquina nativas. El CLR también proporciona otros servicios relacionados con la recolección de elementos no utilizados, el control de excepciones y la administración de recursos. El código que se ejecuta en el CLR se conoce a veces como "código administrado", a diferencia del "código no administrado" que se compila en lenguaje de máquina nativo destinado a un sistema específico. En el siguiente diagrama se ilustran las relaciones de tiempo de compilación y tiempo de ejecución de archivos de código fuente de C#, las bibliotecas de clases de .NET Framework, los ensamblados y el CLR.

La interoperabilidad entre lenguajes es una característica fundamental de .NET Framework. Debido a que el código IL generado por el compilador de C# cumple la especificación de tipo común (CTS), este código puede interactuar con el código generado a partir de las versiones .NET de Visual Basic, Visual C++ o cualquiera de los más de 20 lenguajes compatibles con CTS. Un solo ensamblado puede contener varios módulos escritos en diferentes lenguajes .NET y los tipos se pueden hacer referencia mutuamente igual que si estuvieran escritos en el mismo lenguaje.

Además de los servicios de tiempo de ejecución, .NET Framework también incluye una amplia biblioteca de más de 4000 clases organizadas en espacios de nombres que proporcionan una gran variedad de funciones útiles para todo, desde la entrada y la salida de archivos, pasando por la manipulación de cadenas para el análisis XML, hasta controles de formularios Windows Forms. La aplicación de C# típica usa mucho la biblioteca de clases de .NET Framework para controlar tareas comunes de infraestructura.

Para más información sobre .NET Framework, consulte Introducción a Microsoft .NET Framework.

Vea también

- Introducción a Visual C#

Tutoriales de C#

28/03/2020 • 7 minutes to read • Edit Online

Le damos la bienvenida a los tutoriales de C#. Se empieza con lecciones interactivas que se pueden ejecutar en el explorador. Los tutoriales posteriores y más avanzados facilitan el trabajo con las herramientas de desarrollo de .NET para crear programas C# en su máquina.

Tutoriales interactivos de introducción a C#

Si quiere iniciar la exploración en formato de vídeo, la serie de vídeos C# 101 proporciona una introducción a C#. Aprenderá los conceptos que puede explorar en estos tutoriales.

En las primeras lecciones se explican los conceptos de C# con la utilización de pequeños fragmentos de código. Aprenderá los datos básicos de la sintaxis de C# y cómo trabajar con tipos de datos como cadenas, números y booleanos. Se trata de material totalmente interactivo, que le permitirá empezar a escribir y ejecutar código en cuestión de minutos. En las primeras lecciones se asume que no dispone de conocimientos previos sobre programación o sobre el lenguaje C#.

Hola mundo

En el tutorial Hola mundo, creará el programa de C# más básico. Explorará el tipo `string` y cómo trabajar con texto.

Números en C#

En el tutorial Números en C#, obtendrá información sobre cómo se almacenan los números en los equipos y cómo realizar cálculos con distintos tipos numéricos. Conocerá los datos básicos sobre cómo realizar redondeos y cálculos matemáticos con C#. Este tutorial también está disponible para ejecutarse localmente en su máquina.

En este tutorial se asume que ha completado la lección Hola mundo.

Bifurcaciones y bucles

En el tutorial Ramas y bucles se explican los datos básicos sobre la selección de diferentes rutas de acceso de la ejecución del código en función de los valores almacenados en variables. Aprenderá los datos básicos del flujo de control, es decir, cómo los programas toman decisiones y eligen distintas acciones. Este tutorial también está disponible para ejecutarse localmente en su máquina.

En este tutorial se supone que ha completado las lecciones Hola mundo y Números en C#.

Colección de listas

En la lección Colección de listas se ofrece información general sobre el tipo de colección de listas que almacena secuencias de datos. Se explica cómo agregar y quitar elementos, buscarlos y ordenar las listas. Explorará los diferentes tipos de listas. Este tutorial también está disponible para ejecutarse localmente en su máquina.

En este tutorial se presupone que ha completado las lecciones que se muestran anteriormente.

Introducción a C#: trabajo local

Todos los tutoriales de introducción posteriores a la lección Hola mundo se encuentran disponibles en el entorno

de desarrollo local. Al final de cada tutorial, decida si desea continuar con la siguiente lección en línea o en su propia máquina. Hay vínculos que le ayudarán a configurar el entorno y continuar con el siguiente tutorial en su máquina.

Explorar las nuevas características de C#

- Prueba de las características nuevas en C# 6 de manera interactiva: explore las características agregadas en C# 6 de manera interactiva en el explorador.
- Interpolación de cadenas: muestra cómo usar la interpolación de cadenas para crear cadenas con formato en C#.
- Tipos de referencia que aceptan valores NULL: se muestra cómo usar los tipos de referencia que aceptan valores NULL para expresar la intención de referencias nulas.
- Actualización de un proyecto para usar tipos de referencia nula: demuestra técnicas para actualizar un proyecto existente y usar los tipos de referencia nula.
- Ampliación de las funcionalidades de datos mediante la coincidencia de patrones: demuestra cómo usar la coincidencia de patrones para ampliar los tipos más allá de sus características principales.
- Trabajo con secuencias de datos mediante índices y rangos: Muestra la nueva sintaxis adecuada para tener acceso a elementos o rangos individuales de un contenedor de datos secuencial.

Tutoriales generales

Los siguientes tutoriales le permiten crear programas de C# mediante .NET Core:

- Aplicación de consola: muestra la E/S de consola, la estructura de una aplicación de consola y los aspectos básicos del modelo de programación asincrónica basado en tareas.
- Cliente de REST: muestra las comunicaciones web, la serialización de JSON y las características orientadas a objetos del lenguaje C#.
- Herencia en C# y .NET: muestra la herencia en C#, incluido el uso de la herencia para definir clases base, clases base abstractas y clases derivadas.
- Trabajar con LINQ: muestra muchas de las características de LINQ y los elementos del lenguaje compatibles.
- Uso de atributos: muestra cómo crear y usar atributos en C#.
- En el tutorial Interpolación de cadenas se muestra cómo insertar valores en una cadena. Aprenderá a crear una cadena interpolada con expresiones C# insertadas y a controlar la apariencia del texto de los resultados de expresión en la cadena de resultado. Este tutorial también está disponible para ejecutarse localmente en su máquina.

Introducción a C#

29/06/2020 • 5 minutes to read • Edit Online

Le damos la bienvenida a los tutoriales de introducción a C#. Estas lecciones empiezan con código interactivo que puede ejecutar en su explorador. Puede obtener información sobre los conceptos básicos de C# en la serie de vídeos C# 101 antes de comenzar estas lecciones interactivas.

En las primeras lecciones se explican los conceptos de C# con la utilización de pequeños fragmentos de código. Aprenderá los datos básicos de la sintaxis de C# y cómo trabajar con tipos de datos como cadenas, números y booleanos. Se trata de material totalmente interactivo, que le permitirá empezar a escribir y ejecutar código en cuestión de minutos. En las primeras lecciones se asume que no dispone de conocimientos previos sobre programación o sobre el lenguaje C#.

Puede probar estos tutoriales en entornos diferentes. Los conceptos que aprenderá son los mismos. La diferencia estará en el tipo de experiencia que elija:

- En el explorador, en la plataforma de documentos: esta experiencia inserta una ventana de código de C# ejecutable en las páginas de documentos. Deberá escribir y ejecutar el código de C# en el explorador.
- En la experiencia de Microsoft Learn: esta ruta de aprendizaje contiene varios módulos en los que se exponen los conceptos básicos de C#.
- En Jupyter desde Binder: puede experimentar con código de C# en un cuaderno de Jupyter en Binder.
- En el equipo local: una vez que haya explorado en línea, puede descargar el SDK de .NET Core y compilar programas en su equipo.

Todos los tutoriales de introducción posteriores a la lección Hola mundo se encuentran disponibles mediante la experiencia de explorador en línea o en el entorno de desarrollo local. Al final de cada tutorial, decida si desea continuar con la siguiente lección en línea o en su propia máquina. Hay vínculos que le ayudarán a configurar el entorno y continuar con el siguiente tutorial en su máquina.

Hola mundo

En el tutorial Hola mundo, creará el programa de C# más básico. Explorará el tipo `string` y cómo trabajar con texto. También puede usar la ruta de acceso en Microsoft Learn o en Jupyter desde Binder.

Números en C#

En el tutorial Números en C#, obtendrá información sobre cómo se almacenan los números en los equipos y cómo realizar cálculos con distintos tipos numéricos. Conocerá los datos básicos sobre cómo realizar redondeos y cálculos matemáticos con C#. Este tutorial también está disponible para ejecutarse localmente en su máquina.

En este tutorial se asume que ha completado la lección Hola mundo.

Bifurcaciones y bucles

En el tutorial Ramas y bucles se explican los datos básicos sobre la selección de diferentes rutas de acceso de la ejecución del código en función de los valores almacenados en variables. Aprenderá los datos básicos del flujo de control, es decir, cómo los programas toman decisiones y eligen distintas acciones. Este tutorial también está disponible para ejecutarse localmente en su máquina.

En este tutorial se asume que ha completado las lecciones Hola mundo y Números en C#.

Colección de listas

En la lección Colección de listas se ofrece información general sobre el tipo de colección de listas que almacena secuencias de datos. Se explica cómo agregar y quitar elementos, buscarlos y ordenar las listas. Explorará los diferentes tipos de listas. Este tutorial también está disponible para ejecutarse localmente en su máquina.

En este tutorial se asume que ha completado las lecciones que se muestran anteriormente.

Introducción a las clases

Este tutorial solo se encuentra disponible para ejecutarse en el equipo con un entorno de desarrollo local propio y .NET Core. Creará una aplicación de consola y conocerá las características básicas orientadas a objetos que forman parte del lenguaje C#.

En este tutorial se presupone que ha completado los tutoriales de introducción en línea y ha instalado el SDK de .NET Core y Visual Studio Code.

Familiarización con las herramientas de desarrollo de .NET

18/03/2020 • 3 minutes to read • Edit Online

El primer paso en la ejecución de un tutorial en el equipo es configurar un entorno de desarrollo. El tutorial de .NET Hola mundo en 10 minutos cuenta con instrucciones para configurar el entorno de desarrollo local en Windows, Linux o macOS.

Como alternativa, puede instalar el SDK de .NET Core y Visual Studio Code.

Flujo de desarrollo de aplicaciones básico

Podrá crear aplicaciones mediante el comando `dotnet new`. Este comando genera los archivos y los recursos necesarios para la aplicación. Los tutoriales de introducción a C# usan el tipo de aplicación `console`. Cuando conozca los conceptos básicos, puede expandirlo a otros tipos de aplicaciones.

Los otros comandos que se van a utilizar son `dotnet build` para compilar el archivo ejecutable, y `dotnet run` para ejecutar el archivo ejecutable.

Elección del tutorial

Puede empezar con cualquiera de los siguientes tutoriales:

Números en C#

En el tutorial Números en C#, obtendrá información sobre cómo se almacenan los números en los equipos y cómo realizar cálculos con distintos tipos numéricos. Conocerá los datos básicos sobre cómo realizar redondeos y cálculos matemáticos con C#.

En este tutorial se asume que ha completado la lección Hola mundo.

Bifurcaciones y bucles

En el tutorial Ramas y bucles se explican los datos básicos sobre la selección de diferentes rutas de acceso de la ejecución del código en función de los valores almacenados en variables. Aprenderá los datos básicos del flujo de control, es decir, cómo los programas toman decisiones y eligen distintas acciones.

En este tutorial se supone que ha completado las lecciones Hola mundo y Números en C#.

Colección de listas

En la lección Colección de listas se ofrece información general sobre el tipo de colección de listas que almacena secuencias de datos. Se explica cómo agregar y quitar elementos, buscarlos y ordenar las listas. Explorará los diferentes tipos de listas.

En este tutorial se presupone que ha completado las lecciones que se muestran anteriormente.

Introducción a las clases

Este tutorial final de introducción a C# solo se encuentra disponible para ejecutarse en el equipo con un entorno de desarrollo local propio y .NET Core. Creará una aplicación de consola y conocerá las características básicas

orientadas a objetos que forman parte del lenguaje C#.

Manipular números enteros y de punto flotante en C#

12/05/2020 • 16 minutes to read • Edit Online

En este tutorial se explican los tipos numéricos en C# de manera interactiva. Escribirá pequeñas cantidades de código y luego compilará y ejecutará ese código. El tutorial contiene una serie de lecciones que ofrecen información detallada sobre los números y las operaciones matemáticas en C#. En ellas se enseñan los aspectos básicos del lenguaje C#.

En este tutorial se supone que cuenta con una máquina que puede usar para el desarrollo. El tutorial de .NET Hola mundo en 10 minutos cuenta con instrucciones para configurar el entorno de desarrollo local en Windows, Linux o macOS. En Become familiar with the development tools (Familiarizarse con las herramientas de desarrollo) puede obtener información general sobre los comandos que usará, donde hay vínculos que amplían la información.

Análisis de las operaciones matemáticas con enteros

Cree un directorio denominado *numbers-quickstart*. Conviértalo en el directorio actual y ejecute el siguiente comando:

```
dotnet new console -n NumbersInCSharp -o .
```

Abra *Program.cs* en su editor favorito y reemplace la línea `Console.WriteLine("Hello World!");` por el código siguiente:

```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

Ejecute este código escribiendo `dotnet run` en la ventana de comandos.

Ha visto una de las operaciones matemáticas fundamentales con enteros. El tipo `int` representa un **entero**, que puede ser cero o un número entero positivo o negativo. Use el símbolo `+` para la suma. Otros operadores matemáticos comunes con enteros son:

- `-` para resta
- `*` para multiplicación
- `/` para división

Comience por explorar esas operaciones diferentes. Agregue estas líneas después de la línea que escribe el valor de `c`:

```
// subtraction
c = a - b;
Console.WriteLine(c);

// multiplication
c = a * b;
Console.WriteLine(c);

// division
c = a / b;
Console.WriteLine(c);
```

Ejecute este código escribiendo `dotnet run` en la ventana de comandos.

Si quiere, también puede probar a escribir varias operaciones matemáticas en la misma línea. Pruebe `c = a + b - 12 * 17;` por ejemplo. Se permite la combinación de variables y números constantes.

> **TIP**
> Cuando explore C# o cualquier otro lenguaje de programación, cometerá errores al escribir código. El **compilador** buscará dichos errores y los notificará. Si la salida contiene mensajes de error, revise detenidamente el código de ejemplo y el código de la ventana para saber qué debe corregir. Este ejercicio le ayudará a aprender la estructura del código de C#.

Ha terminado el primer paso. Antes comenzar con la siguiente sección, se va a mover el código actual a un método independiente. Con este paso, resulta más fácil empezar con un nuevo ejemplo. Cambie el nombre del método `Main` por `WorkingWithIntegers` y escriba un método `Main` nuevo que llame a `WorkingWithIntegers`. Cuando termine, el código debe tener un aspecto similar al siguiente:

```
using System;

namespace NumbersInCSharp
{
    class Program
    {
        static void WorkingWithIntegers()
        {
            int a = 18;
            int b = 6;

            // addition
            int c = a + b;
            Console.WriteLine(c);

            // subtraction
            c = a - b;
            Console.WriteLine(c);

            // multiplication
            c = a * b;
            Console.WriteLine(c);

            // division
            c = a / b;
            Console.WriteLine(c);
        }

        static void Main(string[] args)
        {
            WorkingWithIntegers();
        }
    }
}
```

Análisis sobre el orden de las operaciones

Convierta en comentario la llamada a `WorkingWithIntegers()`. De este modo la salida estará menos saturada a medida que trabaje en esta sección:

```
//WorkingWithIntegers();
```

El `//` inicia un **comentario** en C#. Los comentarios son cualquier texto que desea mantener en el código fuente pero que no se ejecuta como código. El compilador no genera ningún código ejecutable a partir de comentarios.

El lenguaje C# define la prioridad de las diferentes operaciones matemáticas con reglas compatibles con las reglas aprendidas en las operaciones matemáticas. La multiplicación y división tienen prioridad sobre la suma y resta. Explórelo mediante la adición del código siguiente a su método `Main` y la ejecución de `dotnet run`:

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

La salida muestra que la multiplicación se realiza antes que la suma.

Puede forzar la ejecución de un orden diferente de las operaciones si la operación o las operaciones que realizó primero se incluyen entre paréntesis. Agregue las líneas siguientes y ejecute de nuevo:

```
d = (a + b) * c;
Console.WriteLine(d);
```

Combine muchas operaciones distintas para indagar más. Agregue algo parecido a las líneas siguientes en la parte inferior del método `Main`. Pruebe `dotnet run` de nuevo.

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

Puede que haya observado un comportamiento interesante de los enteros. La división de enteros siempre genera un entero como resultado, incluso cuando se espera que el resultado incluya un decimal o una parte de una fracción.

Si no ha observado este comportamiento, pruebe el código siguiente al final de su método `Main`.

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

Escriba `dotnet run` de nuevo para ver los resultados.

Antes de continuar, vamos a tomar todo el código que ha escrito en esta sección y a colocarlo en un nuevo método. Llame a ese nuevo método `OrderPrecedence`. Debe escribir algo parecido a esto:

```
using System;

namespace NumbersInCSharp
{
    class Program
    {
        static void WorkingWithIntegers()
        {
            int a = 18;
            int b = 6;

            // addition
            int c = a + b;
            Console.WriteLine(c);

            // subtraction
            c = a - b;
            Console.WriteLine(c);

            // multiplication
            c = a * b;
            Console.WriteLine(c);

            // division
            c = a / b;
            Console.WriteLine(c);
        }

        static void OrderPrecedence()
        {
            int a = 5;
            int b = 4;
            int c = 2;
            int d = a + b * c;
            Console.WriteLine(d);

            d = (a + b) * c;
            Console.WriteLine(d);

            d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
            Console.WriteLine(d);

            int e = 7;
            int f = 4;
            int g = 3;
            int h = (e  + f) / g;
            Console.WriteLine(h);
        }

        static void Main(string[] args)
        {
            WorkingWithIntegers();

            OrderPrecedence();

        }
    }
}
```

Información sobre los límites y la precisión de los enteros

En el último ejemplo se ha mostrado que la división de enteros trunca el resultado. Puede obtener el **resto** con el operador de **módulo**, el carácter `%`. Pruebe el código siguiente en el método `Main`:

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

El tipo de entero de C# difiere de los enteros matemáticos en un aspecto: el tipo `int` tiene límites mínimo y máximo. Agregue este código a su método `Main` para ver esos límites:

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

Si un cálculo genera un valor que supera los límites, se producirá una condición de **subdesbordamiento** o **desbordamiento**. La respuesta parece ajustarse de un límite al otro. Agregue estas dos líneas a su método `Main` para ver un ejemplo:

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

Tenga en cuenta que la respuesta está muy próxima al entero mínimo (negativo). Es lo mismo que `min + 2`. La operación de suma **desbordó** los valores permitidos para los enteros. La respuesta es un número negativo muy grande porque un desbordamiento "se ajusta" desde el valor de entero más alto posible al más bajo.

Hay otros tipos numéricos con distintos límites y precisiones que podría usar si el tipo `int` no satisface sus necesidades. Vamos a explorar ahora esos otros tipos.

Una vez más, vamos a mover el código que escribió en esta sección a un método independiente. Denomínelo `TestLimits`.

Operaciones con el tipo double

El tipo numérico `double` representa números de punto flotante de doble precisión. Puede que no esté familiarizado con estos términos. Un número de **punto flotante** resulta útil para representar números no enteros cuya magnitud puede ser muy grande o pequeña. La **precisión doble** es un término relativo que describe el número de dígitos binarios que se usan para almacenar el valor. Los números de **precisión doble** tienen el doble del número de dígitos binarios que la **precisión sencilla**. En los equipos modernos, es más habitual usar números con precisión doble que con precisión sencilla. Los números de **precisión sencilla** se declaran mediante la palabra clave `float`. Comencemos a explorar. Agregue el siguiente código y observe el resultado:

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

Tenga en cuenta que la respuesta incluye la parte decimal del cociente. Pruebe una expresión algo más complicada con tipos double:

```
double e = 19;
double f = 23;
double g = 8;
double h = (e + f) / g;
Console.WriteLine(h);
```

El intervalo de un valor double es mucho más amplio que en el caso de los valores enteros. Pruebe el código siguiente debajo del que ha escrito hasta ahora:

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

Estos valores se imprimen en notación científica. El número a la izquierda de `E` es la mantisa. El número a la derecha es el exponente, como una potencia de diez.

Al igual que sucede con los números decimales en las operaciones matemáticas, los tipos double en C# pueden presentar errores de redondeo. Pruebe este código:

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

Sabe que repetir `0.3` no es exactamente lo mismo que `1/3`.

Desafío

Pruebe otros cálculos con números grandes, números pequeños, multiplicaciones y divisiones con el tipo `double`. Intente realizar cálculos más complicados.

Una vez que haya invertido un tiempo en ello, tome el código que ha escrito y colóquelo en un nuevo método. Ponga a ese nuevo método el nombre `WorkWithDoubles`.

Trabajo con tipos decimales

Hasta el momento ha visto los tipos numéricos básicos en C#: los tipos integer y double. Pero hay otro tipo más que debe conocer: `decimal`. El tipo `decimal` tiene un intervalo más pequeño, pero mayor precisión que `double`. Observemos lo siguiente:

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Tenga en cuenta que el intervalo es más pequeño que con el tipo `double`. Puede observar una precisión mayor con el tipo decimal si prueba el siguiente código:

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

El sufijo `M` en los números es la forma de indicar que una constante debe usar el tipo `decimal`. De no ser así, el compilador asume el tipo de `double`.

> **NOTE**
> La letra `M` se eligió como la letra más distintiva visualmente entre las palabras clave `double` y `decimal`.

Observe que la expresión matemática con el tipo decimal tiene más dígitos a la derecha del punto decimal.

Desafío

Ahora que ya conoce los diferentes tipos numéricos, escriba código para calcular el área de un círculo cuyo radio sea de 2,50 centímetros. Recuerde que el área de un circulo es igual al valor de su radio elevado al cuadrado multiplicado por Pi. Sugerencia: .NET contiene una constante de Pi, Math.PI, que puede usar para ese valor. Math.PI, al igual que todas las constantes declaradas en el espacio de nombres `System.Math`, es un valor `double`. Por ese motivo, debe usar `double` en lugar de valores `decimal` para este desafío.

Debe obtener una respuesta entre 19 y 20. Puede comprobar la respuesta si consulta el ejemplo de código terminado en GitHub.

Si lo desea, pruebe con otras fórmulas.

Ha completado el inicio rápido "Números en C#". Puede continuar con la guía de inicio rápido Ramas y bucles en su propio entorno de desarrollo.

En estos temas encontrará más información sobre los números en C#:

- Tipos numéricos integrales
- Tipos numéricos de punto flotante
- Conversiones numéricas integradas

Obtenga información sobre la lógica condicional con instrucciones de rama y bucle

29/06/2020 • 17 minutes to read • Edit Online

En este tutorial se enseña a escribir código que analiza variables y cambia la ruta de acceso de ejecución en función de dichas variables. Escriba código de C# y vea los resultados de la compilación y la ejecución. El tutorial contiene una serie de lecciones en las que se analizan las construcciones de bifurcaciones y bucles en C#. En ellas se enseñan los aspectos básicos del lenguaje C#.

En este tutorial se supone que cuenta con una máquina que puede usar para el desarrollo. El tutorial de .NET Hola mundo en 10 minutos cuenta con instrucciones para configurar el entorno de desarrollo local en Windows, Linux o macOS. En Become familiar with the development tools (Familiarizarse con las herramientas de desarrollo) puede obtener información general sobre los comandos que usará, donde hay vínculos que amplían la información.

Toma de decisiones con la instrucción `if`.

Cree un directorio denominado *branches-tutorial*. Conviértalo en el directorio actual y ejecute el siguiente comando:

```
dotnet new console -n BranchesAndLoops -o .
```

Este comando crea una nueva aplicación de consola de .NET Core en el directorio actual.

Abra *Program.cs* en su editor favorito y reemplace la línea `Console.WriteLine("Hello World!");` por el código siguiente:

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

Pruebe este código escribiendo `dotnet run` en la ventana de la consola. Debería ver el mensaje "The answer is greater than 10" (La respuesta es mayor que 10) impreso en la consola.

Modifique la declaración de `b` para que el resultado de la suma sea menor que diez:

```
int b = 3;
```

Escriba `dotnet run` de nuevo. Como la respuesta es menor que diez, no se imprime nada. La **condición** que está probando es false. No tiene ningún código para ejecutar porque solo ha escrito una de las bifurcaciones posibles para una instrucción `if`: la bifurcación true.

> **TIP**
> Cuando explore C# o cualquier otro lenguaje de programación, cometerá errores al escribir código. El compilador buscará dichos errores y los notificará. Fíjese en la salida de error y en el código que generó el error. El error del compilador normalmente puede ayudarle a encontrar el problema.

En este primer ejemplo se muestran la potencia de `if` y los tipos booleanos. Un *booleano* es una variable que puede tener uno de estos dos valores: `true` o `false`. C# define un tipo especial `bool` para las variables booleanas. La instrucción `if` comprueba el valor de `bool`. Cuando el valor es `true`, se ejecuta la instrucción que sigue a `if`. De lo contrario, se omite.

Este proceso de comprobación de condiciones y ejecución de instrucciones en función de esas condiciones es muy eficaz.

Operaciones conjuntas con if y else

Para ejecutar un código distinto en las bifurcaciones true y false, cree una bifurcación `else` para que se ejecute cuando la condición sea false. Pruebe esto. Agregue las dos últimas líneas del código siguiente a su método `Main` (ya debe tener los cuatro primeros):

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

La instrucción que sigue a la palabra clave `else` se ejecuta solo si la condición de prueba es `false`. La combinación de `if` y `else` con condiciones booleanas ofrece toda la eficacia necesaria para administrar una condición `true` y `false` simultáneamente.

> **IMPORTANT**
> La sangría debajo de las instrucciones `if` y `else` se utiliza para los lectores humanos. El lenguaje C# no considera significativos los espacios en blanco ni las sangrías. La instrucción que sigue a la palabra clave `if` o `else` se ejecutará en función de la condición. Todos los ejemplos de este tutorial siguen una práctica común para aplicar sangría a las líneas en función del flujo de control de las instrucciones.

Dado que la sangría no es significativa, debe usar `{` y `}` para indicar si desea que más de una instrucción forme parte del bloque que se ejecuta de forma condicional. Los programadores de C# suelen usar esas llaves en todas las cláusulas `if` y `else`. El siguiente ejemplo es igual que el que acaba de crear. Modifique el código anterior para que coincida con el código siguiente:

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

> **TIP**
> En el resto de este tutorial, todos los ejemplos de código incluyen las llaves, según las prácticas aceptadas.

Puede probar condiciones más complicadas. Agregue el código siguiente a su método `Main` después del código que ha escrito hasta ahora:

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

El símbolo `==` prueba la *igualdad*. Usar `==` permite distinguir la prueba de igualdad de la asignación, que verá en `a = 5`.

`&&` representa "y". Significa que ambas condiciones deben cumplirse para ejecutar la instrucción en la bifurcación true. En estos ejemplos también se muestra que puede tener varias instrucciones en cada bifurcación condicional, siempre que las encierre entre `{` y `}`.

También puede usar `||` para representar "o". Agregue el código siguiente antes del que ha escrito hasta ahora:

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

Modifique los valores de `a`, `b` y `c` y cambie entre `&&` y `||` para explorar. Obtendrá más conocimientos sobre el funcionamiento de los operadores `&&` y `||`.

Ha terminado el primer paso. Antes comenzar con la siguiente sección, se va a mover el código actual a un método independiente. Con este paso, resulta más fácil empezar con un nuevo ejemplo. Cambie el nombre del método `Main` por `ExploreIf` y escriba un método `Main` nuevo que llame a `ExploreIf`. Cuando termine, el código debe tener un aspecto similar al siguiente:

```
using System;

namespace BranchesAndLoops
{
    class Program
    {
        static void ExploreIf()
        {
            int a = 5;
            int b = 3;
            if (a + b > 10)
            {
                Console.WriteLine("The answer is greater than 10");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
            }

            int c = 4;
            if ((a + b + c > 10) && (a > b))
            {
                Console.WriteLine("The answer is greater than 10");
                Console.WriteLine("And the first number is greater than the second");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
                Console.WriteLine("Or the first number is not greater than the second");
            }

            if ((a + b + c > 10) || (a > b))
            {
                Console.WriteLine("The answer is greater than 10");
                Console.WriteLine("Or the first number is greater than the second");
            }
            else
            {
                Console.WriteLine("The answer is not greater than 10");
                Console.WriteLine("And the first number is not greater than the second");
            }
        }

        static void Main(string[] args)
        {
            ExploreIf();
        }
    }
}
```

Convierta en comentario la llamada a `ExploreIf()`. De este modo la salida estará menos saturada a medida que trabaje en esta sección:

```
//ExploreIf();
```

El `//` inicia un **comentario** en C#. Los comentarios son cualquier texto que desea mantener en el código fuente pero que no se ejecuta como código. El compilador no genera ningún código ejecutable a partir de comentarios.

Uso de bucles para repetir las operaciones

En esta sección se usan **bucles** para repetir las instrucciones. Pruebe este código en su método `Main`:

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

La instrucción `while` comprueba una condición y ejecuta la instrucción o el bloque de instrucciones que aparece después de `while`. La comprobación de la condición y la ejecución de dichas instrucciones se repetirán hasta que la condición sea false.

En este ejemplo aparece otro operador nuevo. El código `++` que aparece después de la variable `counter` es el operador de **incremento**. Suma un valor de uno al valor de `counter` y almacena dicho valor en la variable de `counter`.

> **IMPORTANT**
> Asegúrese de que la condición del bucle `while` cambia a false mientras ejecuta el código. En caso contrario, se crea un **bucle infinito** donde nunca finaliza el programa. Esto no está demostrado en este ejemplo, ya que tendrá que forzar al programa a cerrar mediante **CTRL-C** u otros medios.

El bucle `while` prueba la condición antes de ejecutar el código que sigue a `while`. El bucle `do` ... `while` primero ejecuta el código y después comprueba la condición. El bucle *do while* se muestra en el código siguiente:

```
int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

Este bucle `do` y el bucle `while` anterior generan el mismo resultado.

Operaciones con el bucle for

El bucle **for** se utiliza con frecuencia en C#. Pruebe este código en su método Main():

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

El código anterior funciona de la misma forma que los bucles `while` y `do` que ya ha usado. La instrucción `for` consta de tres partes que controlan su funcionamiento.

La primera parte es el **inicializador de for**: `int index = 0;` declara que `index` es la variable de bucle y establece su valor inicial en `0`.

La parte central es la **condición de for**: `index < 10` declara que este bucle `for` debe continuar ejecutándose mientras que el valor del contador sea menor que diez.

La última parte es el **iterador de for**: `index++` especifica cómo modificar la variable de bucle después de ejecutar el bloque que sigue a la instrucción `for`. En este caso, especifica que `index` debe incrementarse en uno cada vez que el bloque se ejecuta.

Experimente usted mismo. Pruebe cada una de las siguientes variaciones:

- Cambie el inicializador para que se inicie en un valor distinto.
- Cambie la condición para que se detenga en un valor diferente.

Cuando haya terminado, escriba algo de código para practicar con lo que ha aprendido.

Hay otra instrucción de bucle que no se trata en este tutorial: la instrucción `foreach`. La instrucción `foreach` repite su instrucción con cada elemento de una secuencia de elementos. Se usa más a menudo con *colecciones*, por lo que se trata en el siguiente tutorial.

Bucles anidados creados

Se puede anidar un bucle `while`, `do` o `for` dentro de otro para crear una matriz mediante la combinación de cada elemento del bucle externo con cada elemento del bucle interno. Vamos a crear un conjunto de pares alfanuméricos para representar filas y columnas.

Un bucle `for` puede generar las filas:

```
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

Otro bucle puede generar las columnas:

```
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

Puede anidar un bucle dentro de otro para formar pares:

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

Puede ver que el bucle externo se incrementa una vez con cada ejecución completa del bucle interno. Invierta el anidamiento de filas y columnas, y vea los cambios por sí mismo.

Combinación de ramas y bucles

Ahora que ya ha obtenido la información sobre el bucle `if` y las construcciones de bucles con el lenguaje C#, trate de escribir código de C# para obtener la suma de todos los enteros de uno a veinte que se puedan dividir entre tres. Tenga en cuenta las siguientes sugerencias:

- El operador `%` proporciona el resto de una operación de división.
- La instrucción `if` genera la condición para saber si un número debe formar parte de la suma.
- El bucle `for` puede facilitar la repetición de una serie de pasos para todos los números comprendidos entre el uno y el veinte.

Pruébelo usted mismo. Después, revise cómo lo ha hecho. Debe obtener 63 como respuesta. Puede ver una

respuesta posible mediante la visualización del código completado en GitHub.

Ha completado el tutorial "Bifurcaciones y bucles".

Puede continuar con el tutorial Matrices y colecciones en su propio entorno de desarrollo.

Puede aprender más sobre estos conceptos en los artículos siguientes:

- Instrucciones If y else
- Instrucción while
- Instrucción do
- Instrucción for

Obtenga información sobre cómo administrar colecciones de datos mediante el tipo de lista genérico

29/06/2020 • 10 minutes to read • Edit Online

En este tutorial de presentación se proporciona una introducción al lenguaje C# y se exponen los conceptos básicos de la clase List<T>.

En este tutorial se supone que cuenta con una máquina que puede usar para el desarrollo. El tutorial de .NET Hola mundo en 10 minutos cuenta con instrucciones para configurar el entorno de desarrollo local en Windows, Linux o macOS. En Familiarización con las herramientas de desarrollo de .NET puede obtener información general sobre los comandos que usará, donde hay vínculos que amplían la información.

Un ejemplo de lista básico

Cree un directorio denominado *list-tutorial*. Conviértalo en el directorio actual y ejecute `dotnet new console`.

Abra *Program.cs* en su editor favorito y reemplace el código existente por el siguiente:

```
using System;
using System.Collections.Generic;

namespace list_tutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            var names = new List<string> { "<name>", "Ana", "Felipe" };
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }
        }
    }
}
```

Reemplace `<name>` por su propio nombre. Guarde *Program.cs*. Escriba `dotnet run` en la ventana de la consola para probarlo.

Ha creado una lista de cadenas, ha agregado tres nombres a esa lista y ha impreso los nombres en MAYÚSCULA. Los conceptos aplicados ya se han aprendido en los tutoriales anteriores para recorrer en bucle la lista.

El código para mostrar los nombres usa la característica interpolación de cadenas. Si un valor de `string` va precedido del carácter `$`, significa que puede insertar código de C# en la declaración de cadena. La cadena real reemplaza a ese código de C# con el valor que genera. En este ejemplo, reemplaza `{name.ToUpper()}` con cada nombre, convertido a mayúsculas, porque se llama al método ToUpper.

Vamos a continuar indagando.

Modificación del contenido de las listas

La colección creada usa el tipo List<T>. Este tipo almacena las secuencias de elementos. Especifique el tipo de los

elementos entre corchetes angulares.

Un aspecto importante de este tipo List<T> es que se puede aumentar o reducir, lo que permite agregar o quitar elementos. Agregue este código antes del corchete de cierre `}` en el método `Main`:

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Se han agregado dos nombres más al final de la lista. También se ha quitado uno. Guarde el archivo y escriba `dotnet run` para probarlo.

List<T> también permite hacer referencia a elementos individuales a través del **índice**. Coloque el índice entre los tokens `[` y `]` después del nombre de la lista. C# utiliza 0 para el primer índice. Agregue este código directamente después del código que acaba de agregar y pruébelo:

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

No se puede acceder a un índice si se coloca después del final de la lista. Recuerde que los índices empiezan en 0, por lo que el índice más grande válido es uno menos que el número de elementos de la lista. Puede comprobar durante cuánto tiempo la lista usa la propiedad Count. Agregue el código siguiente al final del método Main:

```
Console.WriteLine($"The list has {names.Count} people in it");
```

Guarde el archivo y vuelva a escribir `dotnet run` para ver los resultados.

Búsqueda y orden en las listas

En los ejemplos se usan listas relativamente pequeñas, pero las aplicaciones a menudo pueden crear listas que contengan muchos más elementos, en ocasiones, con una numeración que engloba millares. Para encontrar elementos en estas colecciones más grandes, debe buscar diferentes elementos en la lista. El método IndexOf busca un elemento y devuelve su índice. Si el elemento no está en la lista, `IndexOf` devuelve `-1`. Agregue este código en la parte inferior del método `Main`:

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns {index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");

}
```

Los elementos de la lista también se pueden ordenar. El método Sort clasifica todos los elementos de la lista en su orden normal (por orden alfabético si se trata de cadenas). Agregue este código en la parte inferior del método `Main`:

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

Guarde el archivo y escriba `dotnet run` para probar esta última versión.

Antes comenzar con la siguiente sección, se va a mover el código actual a un método independiente. Con este paso, resulta más fácil empezar con un nuevo ejemplo. Cambie el nombre del método `Main` por `WorkingWithStrings` y escriba un método `Main` nuevo que llame a `WorkingWithStrings`. Cuando termine, el código debe tener un aspecto similar al siguiente:

```
using System;
using System.Collections.Generic;

namespace list_tutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            WorkingWithStrings();
        }

        static void WorkingWithStrings()
        {
            var names = new List<string> { "<name>", "Ana", "Felipe" };
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }

            Console.WriteLine();
            names.Add("Maria");
            names.Add("Bill");
            names.Remove("Ana");
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }

            Console.WriteLine($"My name is {names[0]}");
            Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

            Console.WriteLine($"The list has {names.Count} people in it");

            var index = names.IndexOf("Felipe");
            if (index == -1)
            {
                Console.WriteLine($"When an item is not found, IndexOf returns {index}");
            }
            else
            {
                Console.WriteLine($"The name {names[index]} is at index {index}");
            }

            index = names.IndexOf("Not Found");
            if (index == -1)
            {
                Console.WriteLine($"When an item is not found, IndexOf returns {index}");
            }
            else
            {
                Console.WriteLine($"The name {names[index]} is at index {index}");

            }

            names.Sort();
            foreach (var name in names)
            {
                Console.WriteLine($"Hello {name.ToUpper()}!");
            }
        }
    }
}
```

Listas de otros tipos

Hasta el momento, se ha usado el tipo `string` en las listas. Se va a crear una lista List<T> con un tipo distinto. Se va a crear una serie de números.

Agregue lo siguiente en la parte inferior del método `Main` nuevo:

```
var fibonacciNumbers = new List<int> {1, 1};
```

Se crea una lista de enteros y se definen los dos primeros enteros con el valor 1. Son los dos primeros valores de una *sucesión de Fibonacci*, una secuencia de números. Cada número sucesivo de Fibonacci se obtiene con la suma de los dos números anteriores. Agregue este código:

```
var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
    Console.WriteLine(item);
```

Guarde el archivo y escriba `dotnet run` para ver los resultados.

TIP

Para centrarse solo en esta sección, puede comentar el código que llama a `WorkingWithStrings();`. Solo debe colocar dos caracteres `/` delante de la llamada, como en: `// WorkingWithStrings();`.

Desafío

Trate de recopilar los conceptos que ha aprendido en esta lección y en las anteriores. Amplíe lo que ha creado hasta el momento con los números de Fibonacci. Pruebe a escribir el código para generar los veinte primeros números de la secuencia. (Como sugerencia, el 20º número de la serie de Fibonacci es 6765).

Desafío completo

Puede ver un ejemplo de solución en el ejemplo de código terminado en GitHub.

Con cada iteración del bucle, se obtienen los dos últimos enteros de la lista, se suman y se agrega el valor resultante a la lista. El bucle se repite hasta que se hayan agregado veinte elementos a la lista.

Enhorabuena, ha completado el tutorial sobre las listas. Puede continuar con el tutorial Introducción a las clases en su propio entorno de desarrollo.

Puede obtener más información sobre cómo trabajar con el tipo `List` en el artículo de la Guía de .NET que trata sobre las colecciones. Ahí también podrá conocer muchos otros tipos de colecciones.

Exploración de la programación orientada a objetos con clases y objetos

29/06/2020 • 17 minutes to read • Edit Online

En este tutorial se supone que cuenta con una máquina que puede usar para el desarrollo. El tutorial de .NET Hola mundo en 10 minutos cuenta con instrucciones para configurar el entorno de desarrollo local en Windows, Linux o macOS. En Become familiar with the development tools (Familiarizarse con las herramientas de desarrollo) puede obtener información general sobre los comandos que usará, donde hay vínculos que amplían la información.

Creación de una aplicación

En una ventana de terminal, cree un directorio denominado *clases*. Creará la aplicación ahí. Cambie a ese directorio y escriba `dotnet new console` en la ventana de la consola. Este comando crea la aplicación. Abra *Program.cs*. El resultado debería tener un aspecto similar a este:

```
using System;

namespace classes
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

En este tutorial, se van a crear tipos nuevos que representan una cuenta bancaria. Normalmente los desarrolladores definen cada clase en un archivo de texto diferente. De esta forma, la tarea de administración resulta más sencilla a medida que aumenta el tamaño del programa. Cree un archivo denominado *CuentaBancaria.cs* en el directorio *clases*.

Este archivo contendrá la definición de una ***cuenta bancaria***. La programación orientada a objetos organiza el código mediante la creación de tipos en forma de ***clases***. Estas clases contienen el código que representa una entidad específica. La clase `BankAccount` representa una cuenta bancaria. El código implementa operaciones específicas a través de métodos y propiedades. En este tutorial, la cuenta bancaria admite el siguiente comportamiento:

1. Tiene un número de diez dígitos que identifica la cuenta bancaria de forma única.
2. Tiene una cadena que almacena el nombre o los nombres de los propietarios.
3. Se puede consultar el saldo.
4. Acepta depósitos.
5. Acepta reintegros.
6. El saldo inicial debe ser positivo.
7. Los reintegros no pueden resultar en un saldo negativo.

Definición del tipo de cuenta bancaria

Puede empezar por crear los datos básicos de una clase que define dicho comportamiento. Cree un archivo con el comando **File:New**. Asígnele el nombre *BankAccount.cs*. Agregue el código siguiente al archivo *BankAccount.cs*:

```
using System;

namespace classes
{
    public class BankAccount
    {
        public string Number { get; }
        public string Owner { get; set; }
        public decimal Balance { get; }

        public void MakeDeposit(decimal amount, DateTime date, string note)
        {
        }

        public void MakeWithdrawal(decimal amount, DateTime date, string note)
        {
        }
    }
}
```

Antes de avanzar, se va a dar un repaso a lo que ha compilado. La declaración `namespace` permite organizar el código de forma lógica. Este tutorial es relativamente pequeño, por lo que deberá colocar todo el código en un espacio de nombres.

`public class BankAccount` define la clase o el tipo que va a crear. Todo lo que se encuentra entre `{` y `}` después de la declaración de clase define el estado y el comportamiento de la clase. Hay cinco ***miembros*** de la clase `BankAccount`. Los tres primeros son ***propiedades***. Las propiedades son elementos de datos que pueden contener código que exige la validación u otras reglas. Los dos últimos son ***métodos***. Los métodos son bloques de código que realizan una única función. La lectura de los nombres de cada miembro debe proporcionar suficiente información tanto al usuario como a otro desarrollador para entender cuál es la función de la clase.

Apertura de una cuenta nueva

La primera característica que se va a implementar es la apertura de una cuenta bancaria. Cuando un cliente abre una cuenta, debe proporcionar un saldo inicial y la información sobre el propietario o los propietarios de esa cuenta.

La creación de un objeto del tipo `BankAccount` conlleva definir un ***constructor*** que asigne dichos valores. Un ***constructor*** es un miembro que tiene el mismo nombre que la clase. Se utiliza para inicializar los objetos de ese tipo de clase. Agregue el siguiente constructor al tipo `BankAccount`. Coloque el siguiente código encima de la declaración de `MakeDeposit`.

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

A los constructores se les llama cuando se crea un objeto mediante `new`. Reemplace la línea `Console.WriteLine("Hello World!");` de *Program.cs* por la siguiente línea (reemplace `<name>` por su nombre):

```
var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with {account.Balance} initial
balance.");
```

Vamos a ejecutar lo que se ha creado hasta ahora. Si usa Visual Studio, seleccione **Iniciar sin depurar** en el menú **Ejecutar**. Si va a usar una línea de comandos, escriba `dotnet run` en el directorio en el que ha creado el proyecto.

¿Ha observado que el número de cuenta está en blanco? Es el momento de solucionarlo. El número de cuenta debe asignarse cuando se construye el objeto. Sin embargo, el autor de la llamada no es el responsable de crearlo. El código de la clase `BankAccount` debe saber cómo asignar nuevos números de cuenta. Una manera sencilla de hacerlo es empezar con un número de diez dígitos. Increméntelo cuando cree cada cuenta. Por último, almacene el número de cuenta actual cuando se construya un objeto.

Agregue una declaración de miembro a la clase `BankAccount`. Coloque la siguiente línea de código después de la llave de apertura `{` al principio de la clase `BankAccount`:

```
private static int accountNumberSeed = 1234567890;
```

Se trata de un miembro de datos. Tiene el estado `private`, lo que significa que solo se puede acceder a él con el código incluido en la clase `BankAccount`. Es una forma de separar las responsabilidades públicas (como tener un número de cuenta) de la implementación privada (cómo se generan los números de cuenta). También es `static`, lo que significa que se comparte entre todos los objetos `BankAccount`. El valor de una variable no estática es único para cada instancia del objeto `BankAccount`. Agregue las dos líneas siguientes al constructor para asignar el número de cuenta: Colóquelas después de la línea donde pone `this.Balance = initialBalance`:

```
this.Number = accountNumberSeed.ToString();
accountNumberSeed++;
```

Escriba `dotnet run` para ver los resultados.

Creación de depósitos y reintegros

La clase de la cuenta bancaria debe aceptar depósitos y reintegros para que el funcionamiento sea adecuado. Se van a implementar depósitos y reintegros con la creación de un diario de cada transacción de la cuenta. Ofrece algunas ventajas con respecto al mero hecho de actualizar el saldo en cada transacción. El historial se puede utilizar para auditar todas las transacciones y administrar los saldos diarios. Con el cálculo del saldo a partir del historial de todas las transacciones, cuando proceda, todos los errores de una única transacción que se solucionen se reflejarán correctamente en el saldo cuando se realice el siguiente cálculo.

Se va a empezar por crear un tipo para representar una transacción. Se trata de un tipo simple que no tiene ninguna responsabilidad. Necesita algunas propiedades. Cree un archivo denominado *Transaction.cs*. Agregue el código siguiente a él:

```
using System;

namespace classes
{
    public class Transaction
    {
        public decimal Amount { get; }
        public DateTime Date { get; }
        public string Notes { get; }

        public Transaction(decimal amount, DateTime date, string note)
        {
            this.Amount = amount;
            this.Date = date;
            this.Notes = note;
        }
    }
}
```

Ahora se va a agregar List<T> de objetos `Transaction` a la clase `BankAccount`. Agregue la siguiente declaración después del constructor en el archivo *BankAccount.cs*:

```
private List<Transaction> allTransactions = new List<Transaction>();
```

La clase List<T> requiere la importación de un espacio de nombres diferente. Agregue lo siguiente al principio de *CuentaBancaria.cs*:

```
using System.Collections.Generic;
```

Ahora se va a cambiar la forma en que se notifica `Balance`. Esto se puede conseguir con la suma de los valores de todas las transacciones. Modifique la declaración de `Balance` en la clase `BankAccount` por lo siguiente:

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

En este ejemplo se muestra un aspecto importante de las ***propiedades***. Ahora va a calcular el saldo cuando otro programador solicite el valor. El cálculo enumera todas las transacciones y proporciona la suma como el saldo actual.

Después, implemente los métodos `MakeDeposit` y `MakeWithdrawal`. Estos métodos exigirán las dos reglas finales: el saldo inicial debe ser positivo y ningún reintegro debe generar un saldo negativo.

Esta operación introduce el concepto de las ***excepciones***. La forma habitual de indicar que un método no puede completar su trabajo correctamente consiste en generar una excepción. El tipo de excepción y el mensaje asociado a ella describen el error. En este caso, el método `MakeDeposit` genera una excepción si el importe del depósito es negativo. El método `MakeWithdrawal` genera una excepción si la cantidad retirada es negativa o si la aplicación del

reintegro tiene como resultado un saldo negativo: Agregue el código siguiente después de la declaración de la lista `allTransactions`:

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}
```

La instrucción `throw` **genera** una excepción. La ejecución del bloque actual finaliza y el control se transfiere al primer bloque `catch` coincidente que se encuentra en la pila de llamadas. Se agregará un bloque `catch` para probar este código un poco más adelante.

El constructor debe obtener un cambio para que agregue una transacción inicial, en lugar de actualizar el saldo directamente. Puesto que ya escribió el método `MakeDeposit`, llámelo desde el constructor. El constructor terminado debe tener este aspecto:

```
public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now es una propiedad que devuelve la fecha y hora actuales. Para probar esto, agregue algunos depósitos y reintegros en el método `Main`, a continuación del código que crea un elemento `BankAccount`:

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Después, compruebe si detecta las condiciones de error al tratar de crear una cuenta con un saldo negativo. Agregue el código siguiente después del código anterior que acaba de agregar:

```
// Test that the initial balances must be positive.
try
{
    var invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
}
```

Use las instrucciones `try` y `catch` para marcar un bloque de código que puede generar excepciones y para detectar los errores que se esperan. Puede usar la misma técnica para probar el código que genera una excepción para un saldo negativo. Agregue el código siguiente al final del método `Main`:

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Guarde el archivo y escriba `dotnet run` para probarlo.

Desafío: registro de todas las transacciones

Para finalizar este tutorial, puede escribir el método `GetAccountHistory` que crea `string` para el historial de transacciones. Agregue este método al tipo `BankAccount`:

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in allTransactions)
    {
        balance += item.Amount;
        report.AppendLine($"{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

Usa la clase StringBuilder para dar formato a una cadena que contiene una línea para cada transacción. Se ha visto anteriormente en estos tutoriales el código utilizado para dar formato a una cadena. Un carácter nuevo es `\t`. Inserta una pestaña para dar formato a la salida.

Agregue esta línea para probarla en *Program.cs*:

```
Console.WriteLine(account.GetAccountHistory());
```

Ejecute el programa para ver los resultados.

Pasos siguientes

Si se ha quedado bloqueado, puede consultar el origen de este tutorial en el repositorio de GitHub.

Enhorabuena, ha completado todos nuestros tutoriales de introducción a C#. Si le interesa obtener más información, continúe con más tutoriales.

Uso de la interpolación de cadenas para construir cadenas con formato

11/05/2020 • 15 minutes to read • Edit Online

En este tutorial se muestra cómo usar la interpolación de cadenas de C# para insertar valores en una cadena de resultado única. Escriba código de C# y vea los resultados de la compilación y la ejecución. Este tutorial contiene una serie de lecciones para mostrarle cómo insertar valores en una cadena y dar formato a estos valores de maneras diferentes.

En este tutorial se supone que cuenta con una máquina que puede usar para el desarrollo. El tutorial de .NET Hola mundo en 10 minutos cuenta con instrucciones para configurar el entorno de desarrollo local en Windows, Linux o macOS. También puede completar la versión interactiva de este tutorial en el explorador.

Crear una cadena interpolada

Cree un directorio denominado *interpolated*. Conviértalo en el directorio actual y ejecute este comando desde una ventana de consola:

```
dotnet new console
```

Este comando crea una nueva aplicación de consola de .NET Core en el directorio actual.

Abra *Program.cs* en su editor favorito y reemplace la línea `Console.WriteLine("Hello World!");` por el código siguiente, teniendo en cuenta que debe reemplazar `<name>` con su nombre:

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Pruebe este código escribiendo `dotnet run` en la ventana de la consola. Al ejecutar el programa, se muestra una cadena única que incluye su nombre en el saludo. La cadena que se incluye en la llamada al método WriteLine es una *expresión de cadena interpolada*. Es un tipo de plantilla que permite construir una sola cadena (denominada *cadena de resultado*) a partir de una cadena que incluye código incrustado. Las cadenas interpoladas son especialmente útiles para insertar valores en una cadena o en cadenas concatenadas (unidas entre sí).

Este sencillo ejemplo contiene los dos elementos que debe tener cada cadena interpolada:

- Un literal de cadena que empieza con el carácter `$` antes del carácter de comillas de apertura. No puede haber ningún espacio entre el símbolo `$` y el carácter de comillas. (Si quiere saber qué pasa si incluye una, inserte un espacio después del carácter `$`, guarde el archivo y vuelva a ejecutar el programa escribiendo `dotnet run` en la ventana de la consola. El compilador de C# muestra un mensaje de error: "error CS1056: carácter no esperado '$'").
- Una o varias *expresiones de interpolación*. Una expresión de interpolación se indica mediante una llave de apertura y de cierre (`{` y `}`). Puede colocar cualquier expresión de C# que devuelva un valor (incluido `null`) dentro de las llaves.

Probemos algunos ejemplos más de interpolación de cadenas con otros tipos de datos.

Incluir diferentes tipos de datos

En la sección anterior, se ha usado una interpolación de cadena para insertar una cadena dentro de otra, pero el resultado de una expresión de interpolación puede ser cualquier tipo de datos. Vamos a incluir valores de distintos tipos de datos en una cadena interpolada.

En el ejemplo siguiente, primero se define un tipo de datos de clase `Vegetable` que tiene una propiedad `Name` y un método `ToString` que reemplaza el comportamiento del método Object.ToString(). El `public` modificador de acceso pone ese método a disposición de cualquier código de cliente para obtener la representación de la cadena de una instancia de `Vegetable`. En el ejemplo, el método `Vegetable.ToString` devuelve el valor de la propiedad `Name` que se inicializa en el constructor `Vegetable`:

```
public Vegetable(string name) => Name = name;
```

Luego se crea una instancia de la clase `Vegetable` denominada `item` al usar el operador `new` y al proporcionar un parámetro de nombre para el constructor `Vegetable`:

```
var item = new Vegetable("eggplant");
```

Por último, se incluye la variable `item` en una cadena interpolada que también contiene un valor DateTime, un valor Decimal y un valor de enumeración `Unit`. Reemplace todo el código de C# en el editor con el código siguiente y, después, use el comando `dotnet run` para ejecutarlo:

```
using System;

public class Vegetable
{
   public Vegetable(string name) => Name = name;

   public string Name { get; }

   public override string ToString() => Name;
}

public class Program
{
   public enum Unit { item, kilogram, gram, dozen };

   public static void Main()
   {
      var item = new Vegetable("eggplant");
      var date = DateTime.Now;
      var price = 1.99m;
      var unit = Unit.item;
      Console.WriteLine($"On {date}, the price of {item} was {price} per {unit}.");
   }
}
```

Observe que la expresión de interpolación `item` de la cadena interpolada se resuelve en el texto "eggplant" en la cadena de resultado. Esto se debe a que, cuando el tipo del resultado de la expresión no es una cadena, el resultado se resuelve en una cadena de la siguiente manera:

- Si la expresión de interpolación se evalúa en `null`, se usa una cadena vacía ("", o String.Empty).
- Si la expresión de interpolación no se evalúa en `null`, se suele llamar al método `ToString` del tipo de resultado. Puede probar esto mediante la actualización de la implementación del método `Vegetable.ToString`. Podría incluso no implementar el método `ToString`, puesto que cada tipo de datos tiene alguna implementación de este método. Para probar esto, comente la definición del método `Vegetable.ToString` del ejemplo (para ello, coloque delante un símbolo de comentario `//`). En el resultado,

se reemplaza la cadena "eggplant" por el nombre del tipo completo ("Vegetable" en este ejemplo), que es el comportamiento predeterminado del método Object.ToString(). El comportamiento predeterminado del método `ToString` para un valor de enumeración es devolver la representación de cadena del valor.

En el resultado de este ejemplo, la fecha es demasiado precisa (el precio de "eggplant" no varía por segundos) y el valor del precio no indica una unidad de moneda. En la sección siguiente se aprende a corregir esos problemas al controlar el formato de representaciones de cadena de los resultados de la expresión.

Control del formato de las expresiones de interpolación

En la sección anterior, las dos cadenas con formato incorrecto se insertaron en la cadena de resultado. Una era un valor de fecha y hora en la que solo la fecha era apropiada. La segunda era un precio que no indicaba su unidad de moneda. Ambos problemas se podían solucionar fácilmente. La interpolación de cadena permite especificar *cadenas de formato* que controlan el formato de tipos específicos. Modifique la llamada a `Console.WriteLine` del ejemplo anterior para incluir las cadenas de formato para las expresiones de fecha y precio, como se muestra en la siguiente línea:

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per {unit}.");
```

Especifique una cadena de formato al colocar dos puntos (":") después de la expresión de interpolación y la cadena de formato. "d" es una cadena de formato de fecha y hora estándar que representa el formato de fecha corta. "C2" es una cadena de formato numérico estándar que representa un número como un valor de moneda con dos dígitos después del separador decimal.

Una serie de tipos de las bibliotecas de .NET admiten un conjunto predefinido de cadenas de formato. Esto incluye todos los tipos numéricos y los tipos de fecha y hora. Para obtener una lista completa de los tipos que admiten cadenas de formato, vea Dar formato a cadenas y tipos de biblioteca de clase .NET en el artículo Aplicar formato a tipos de .NET.

Pruebe a modificar las cadenas de formato en el editor de texto y, cada vez que realice un cambio, vuelva a ejecutar el programa para ver cómo los cambios afectan al formato de fecha y hora y al valor numérico. Cambie "d" en `{date:d}` a "t" (para mostrar el formato de hora corta), "y" (para mostrar el año y el mes) y "yyyy" (para mostrar el año como un número de cuatro dígitos). Cambie "C2" en `{price:C2}` a "e" (para la notación exponencial) y "F3" (para un valor numérico con tres dígitos después del separador decimal).

Además de controlar el formato, también puede controlar el ancho de campo y la alineación de las cadenas con formato incluidas en la cadena de resultado. En la siguiente sección aprenderá a hacerlo.

Control el ancho de campo y la alineación de expresiones de interpolación

Normalmente, cuando el resultado de una expresión de interpolación tiene formato de cadena, esa cadena se incluye en una cadena de resultado sin espacios iniciales ni finales. Especialmente cuando se trabaja con un conjunto de datos, poder controlar el ancho de un campo y la alineación del texto ayuda a generar una salida más legible. Para ver esto, reemplace todo el código en el editor de texto con el código siguiente, y luego escriba `dotnet run` para ejecutar el programa:

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
      var titles = new Dictionary<string, string>()
      {
          ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
          ["London, Jack"] = "Call of the Wild, The",
          ["Shakespeare, William"] = "Tempest, The"
      };

      Console.WriteLine("Author and Title List");
      Console.WriteLine();
      Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
      foreach (var title in titles)
         Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
   }
}
```

Los nombres de los autores están alineados a la izquierda y los títulos que escribieron están alineados a la derecha. Para especificar la alineación, se agrega una coma (",") después de una expresión de interpolación y se designa el ancho de campo *mínimo*. Si el valor especificado es un número positivo, el campo se alinea a la derecha. Si es un número negativo, el campo se alinea a la izquierda.

Pruebe a quitar el signo negativo del código `{"Author",-25}` y `{title.Key,-25}`, y vuelva a ejecutar el ejemplo, como hace este código:

```
Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
   Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");
```

Esta vez, la información del autor está alineada a la derecha.

Puede combinar un especificador de alineación y una cadena de formato en una única expresión de interpolación. Para ello, especifique primero la alineación, seguida de dos puntos y la cadena de formato. Reemplace todo el código dentro del método `Main` con el código siguiente, que muestra tres cadenas con formato con los anchos de campo definidos. Después, ejecute el programa escribiendo el comando `dotnet run`.

```
Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}] [{1063.342,15:N2}] feet");
```

El resultado tiene un aspecto similar a este:

```
[04/14/2018          ] Hour [16        ] [       1,063.34] feet
```

Ha completado el tutorial sobre interpolación de cadenas.

Para más información, vea el tema Interpolación de cadenas y el tutorial Interpolación de cadenas en C#.

Interpolación de cadenas en C#

20/05/2020 • 10 minutes to read • Edit Online

En este tutorial se explica cómo usar la interpolación de cadenas para dar formato a resultados de expresión e incluirlos en una cadena de resultado. En los ejemplos se da por hecho que ya está familiarizado con los conceptos básicos de C# y el formato de tipos .NET. Si no conoce la interpolación de cadenas o el formato de tipos .NET, vea antes el tutorial de interpolación de cadenas interactivo. Para más información sobre cómo aplicar formato a tipos .NET, vea el tema Aplicar formato a tipos en .NET.

> **NOTE**
>
> Los ejemplos de C# de este artículo se ejecutan en el ejecutor de código en línea y área de juegos de Try.NET. Haga clic en el botón **Ejecutar** para ejecutar un ejemplo en una ventana interactiva. Una vez que se ejecuta el código, puede modificar y ejecutar el código modificado si vuelve a hacer clic en **Ejecutar**. El código modificado se ejecuta en la ventana interactiva o, si se produce un error en la compilación, en la ventana interactiva se muestran todos los mensajes de error del compilador de C#.

Introducción

La característica de interpolación de cadenas se basa en la característica de formato compuesto y proporciona una sintaxis más legible y cómoda para incluir resultados de expresiones con formato en una cadena de resultado.

Para distinguir un literal de cadena como una cadena interpolada, antepóngale el símbolo `$`. Puede insertar cualquier expresión de C# válida que devuelva un valor en una cadena interpolada. En el siguiente ejemplo, en cuanto la expresión se evalúa, su resultado se convierte en una cadena y se incluye en una cadena de resultado:

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of {a} and {b} is
{CalculateHypotenuse(a, b)}");

double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 + leg2 * leg2);

// Expected output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

Como se ilustra en el ejemplo, para incluir una expresión en una cadena interpolada hay que meterla entre llaves:

```
{<interpolationExpression>}
```

Las cadenas interpoladas admiten todas las funcionalidades de la característica de formato compuesto de cadena. Esto las convierte en una alternativa más legible al uso del método String.Format.

Cómo especificar una cadena de formato para una expresión de interpolación

Para especificar una cadena de formato compatible con el tipo de resultado de expresión, coloque después de la expresión de interpolación dos puntos (":") y la cadena de formato:

```
{<interpolationExpression>:<formatString>}
```

En el siguiente ejemplo se muestra cómo especificar cadenas de formato estándar y personalizadas para expresiones que generan resultados numéricos o de fecha y hora:

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote
{Math.E:F5} in a letter to Christian Goldbach.");

// Expected output:
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to
Christian Goldbach.
```

Para más información, vea la sección Format String (Componente) del tema Formatos compuestos. En esa sección encontrará vínculos a temas en los que se describen las cadenas de formato estándar y personalizadas compatibles con los tipos base .NET.

Cómo controlar el ancho de campo y la alineación de las expresiones de interpolación con formato

Para especificar el ancho de campo mínimo y la alineación del resultado de expresión con formato, coloque después de la expresión de interpolación una coma (",") y la expresión constante:

```
{<interpolationExpression>,<alignment>}
```

Si el valor de *alignment* es positivo, el resultado de la expresión con formato se alineará a la derecha y, si es negativo, lo hará a la izquierda.

En caso de que haya que especificar una alineación y una cadena de formato, comience por el componente de alineación:

```
{<interpolationExpression>,<alignment>:<formatString>}
```

En el siguiente ejemplo se muestra cómo especificar la alineación y se emplean caracteres de barra vertical ("|") para delimitar los campos de texto:

```
const int NameAlignment = -9;
const int ValueAlignment = 7;

double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a + b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a * b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 / b),ValueAlignment:F3}|");

// Expected output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic|  3.500|
// |Geometric|  3.464|
// |Harmonic |  3.429|
```

Tal y como refleja la salida del ejemplo, si la longitud del resultado de expresión con formato supera el ancho de campo especificado, se omitirá el valor de *alignment*.

Para más información, vea la sección Alignment (Componente) del tema Formatos compuestos.

Cómo usar secuencias de escape en una cadena interpolada

Las cadenas interpoladas admiten todas las secuencias de escape que se usan en los literales de cadena ordinarios. Para más información, vea Secuencias de escape de cadena.

Para interpretar las secuencias de escape literalmente, use un literal de cadena textual. Las cadenas textuales interpoladas comienzan por el carácter `$`, seguido del carácter `@`. A partir C# 8.0, puede usar los tokens `$` y `@` en cualquier orden; tanto `$@"..."` como `@$"..."` son cadenas textuales interpoladas válidas.

Para incluir una llave ("{" o "}") en una cadena de resultado, use dos llaves ("{{" o "}}"). Para más información, vea la sección Llaves de escape del tema Formatos compuestos.

En el siguiente ejemplo se muestra cómo incluir llaves en una cadena de resultado y cómo construir una cadena interpolada textual:

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}} and {{{string.Join(", ",ys)}}}
sets.");

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

Cómo usar un operador condicional ternario `?:` en una expresión de interpolación

Dado que los dos puntos (:) tienen un significado especial en un elemento con una expresión de interpolación, para poder usar un operador condicional en una expresión de este tipo habrá que ponerlo entre paréntesis, como vemos en el siguiente ejemplo:

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" : "tails")}");
}
```

Cómo crear una cadena de resultado específica de la referencia cultural con interpolación de cadenas

Las cadenas interpoladas usan de forma predeterminada la referencia cultural definida actualmente por la propiedad CultureInfo.CurrentCulture en todas las operaciones de formato. Use la conversión implícita de una cadena interpolada en una instancia de System.FormattableString y llame al método ToString(IFormatProvider) correspondiente para crear una cadena de resultado específica de referencia cultural. En el siguiente ejemplo se muestra cómo hacerlo:

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};

var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US       5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL        17-05-18 15:44:55      31.415.926,536
//            05/17/2018 15:44:55      31,415,926.536
```

Tal y como se muestra en el ejemplo, se puede usar una instancia de FormattableString para generar varias cadenas de resultado para varias referencias culturales.

Cómo crear una cadena de resultado usando la referencia de cultura invariable

Puede usar el método estático FormattableString.Invariant junto con el método FormattableString.ToString(IFormatProvider) para resolver una cadena interpolada en una cadena de resultado de InvariantCulture. En el siguiente ejemplo se muestra cómo hacerlo:

```
string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture:
{DateTime.Now}");
Console.WriteLine(messageInInvariantCulture);

// Expected output is like:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Conclusión

En este tutorial se han descrito escenarios habituales en los que se usa la interpolación de cadenas. Para más información sobre la interpolación de cadenas, vea el tema Interpolación de cadenas. Para más información sobre cómo aplicar formato a tipos .NET, vea los temas Aplicar formato a tipos en .NET y Formatos compuestos.

Vea también

- String.Format
- System.FormattableString
- System.IFormattable
- Cadenas

Tutorial: Actualización de las interfaces mediante el uso de métodos de interfaz predeterminados en C# 8.0

29/06/2020 • 11 minutes to read • Edit Online

A partir de C# 8.0 en .NET Core 3.0, puede definir una implementación cuando declare un miembro de una interfaz. El escenario más común es agregar de forma segura a miembros a una interfaz ya publicada y utilizada por clientes incontables.

En este tutorial aprenderá lo siguiente:

- Extender interfaces de forma segura mediante la adición de métodos con implementaciones.
- Crear implementaciones con parámetros para proporcionar mayor flexibilidad.
- Permitir que los implementadores proporcionen una implementación más específica en forma de una invalidación.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core, incluido el compilador de C# 8.0. El compilador de C# 8.0 está disponible a partir de la versión 16.3 de Visual Studio 2019 o del SDK de .NET Core 3.0.

Información general del escenario

Este tutorial comienza con la versión 1 de una biblioteca de relaciones con clientes. Puede obtener la aplicación de inicio en nuestro repositorio de ejemplo en GitHub. La empresa que creó esta biblioteca se dirigía a los clientes con aplicaciones existentes para adoptar sus bibliotecas. Proporcionan definiciones de una interfaz mínima para que la implemente los usuarios de su biblioteca. Esta es la definición de interfaz para un cliente:

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

Definieron una segunda interfaz que representa un pedido:

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

En esas interfaces, el equipo pudo generar una biblioteca para sus usuarios con el fin de crear una mejor experiencia para los clientes. Su objetivo era crear una relación más estrecha con los clientes existentes y mejorar sus relaciones con los clientes nuevos.

Ahora, es momento de actualizar la biblioteca para la próxima versión. Una de las características solicitadas permite un descuento por fidelidad para los clientes que tienen muchos pedidos. Este nuevo descuento por fidelidad se aplica cada vez que un cliente realiza un pedido. El descuento específico es una propiedad de cada cliente individual. Cada implementación de `ICustomer` puede establecer reglas diferentes para el descuento por fidelidad.

La forma más natural para agregar esta funcionalidad es mejorar la interfaz `ICustomer` con un método para aplicar los descuentos por fidelización. Esta sugerencia de diseño es motivo de preocupación entre los desarrolladores experimentados: "Las interfaces son inmutables una vez que se han publicado. ¡Es un cambio importante!" C# 8.0 agrega *las implementaciones de interfaz predeterminadas* para actualizar las interfaces. Los autores de bibliotecas pueden agregar a nuevos miembros a la interfaz y proporcionar una implementación predeterminada para esos miembros.

Las implementaciones de interfaces predeterminadas permiten a los desarrolladores actualizar una interfaz mientras siguen permitiendo que los implementadores invaliden esa implementación. Los usuarios de la biblioteca pueden aceptar la implementación predeterminada como un cambio sin importancia. Si sus reglas de negocios son diferentes, se puede invalidar.

Actualización con los métodos de interfaz predeterminados

El equipo estuvo de acuerdo en la más probable implementación predeterminada: un descuento por fidelidad para los clientes.

La actualización debe proporcionar la funcionalidad para establecer dos propiedades: el número de pedidos necesario para poder recibir el descuento y el porcentaje del descuento. Esto lo convierte en un escenario perfecto para los métodos de interfaz predeterminados. Puede agregar un método a la interfaz de `ICustomer` y proporcionar la implementación más probable. Todas las implementaciones existentes y nuevas pueden usar la implementación predeterminada o proporcionar una propia.

En primer lugar, agregue el nuevo método a la interfaz, incluido su cuerpo:

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

El autor de la biblioteca escribió una primera prueba para comprobar la implementación:

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(1012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2103, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Tenga en cuenta la siguiente parte de la prueba:

```
// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

La conversión de `SampleCustomer` a `ICustomer` es necesaria. La clase `SampleCustomer` no necesita proporcionar una implementación para `ComputeLoyaltyDiscount`. La interfaz `ICustomer` la proporciona. Sin embargo, la clase `SampleCustomer` no hereda miembros de sus interfaces. Esa no ha cambiado. Para poder llamar a cualquier método declarado e implementado en la interfaz, la variable debe ser del tipo de la interfaz: `ICustomer` en este ejemplo.

Proporcionar parametrización

Ese es un buen inicio. Pero la implementación predeterminada es demasiado restrictiva. Muchos consumidores de este sistema pueden elegir diferentes umbrales para el número de compras, una longitud diferente de la pertenencia o un porcentaje diferente del descuento. Puede proporcionar una mejor experiencia de actualización para más clientes proporcionando una manera de establecer esos parámetros. Vamos a agregar un método estático que establezca esos tres parámetros controlando la implementación predeterminada:

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

Hay muchas funcionalidades nuevas de lenguaje que se muestran en este pequeño fragmento de código. Las interfaces ahora pueden incluir miembros estáticos, incluidos campos y métodos. También están habilitados diferentes modificadores de acceso. Los campos adicionales son privados, el nuevo método es público. Todos los modificadores están permitidos en los miembros de la interfaz.

Las aplicaciones que usan la fórmula general para calcular el descuento por fidelidad, pero diferentes parámetros, no necesitan proporcionar una implementación personalizada: pueden establecer los argumentos a través de un método estático. Por ejemplo, el siguiente código establece una "apreciación de cliente" que recompensa a cualquier cliente con más de una pertenencia al mes:

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Extender la implementación predeterminada

El código que ha agregado hasta ahora ha proporcionado una implementación adecuada para los escenarios donde los usuarios quieren algo similar a la implementación predeterminada, o para proporcionar un conjunto de reglas no relacionado. Para una característica final, vamos a refactorizar el código un poco para habilitar los escenarios donde es posible que los usuarios deseen crear en la implementación predeterminada.

Considere una startup que desea captar nuevos clientes. Ofrecen un descuento del 50 % en el primer pedido de un cliente nuevo. De lo contrario, los clientes existentes obtienen el descuento estándar. El autor de la biblioteca necesita mover la implementación predeterminada a un método `protected static` para que cualquier clase que implemente esta interfaz pueda reutilizar el código en su implementación. La implementación predeterminada del miembro de la interfaz llama a este método compartido así:

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

En una implementación de una clase que implementa esta interfaz, la invalidación puede llamar al método auxiliar estático y ampliar esa lógica para proporcionar el descuento de "cliente nuevo":

```
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

Puede ver todo el código terminado en nuestro repositorio de ejemplos en GitHub. Puede obtener la aplicación de inicio en nuestro repositorio de ejemplo en GitHub.

Estas nuevas características significan que las interfaces se pueden actualizar de forma segura cuando hay una implementación predeterminada razonable para esos nuevos miembros. Diseñe cuidadosamente las interfaces para expresar ideas funcionales únicas que puedan implementarse con varias clases. Esto facilita la actualización de esas definiciones de interfaz cuando se descubren nuevos requisitos para esa misma idea funcional.

Tutorial: Funcionalidad de combinación al crear clases mediante interfaces con métodos de interfaz predeterminados

29/06/2020 • 15 minutes to read • Edit Online

A partir de C# 8.0 en .NET Core 3.0, puede definir una implementación cuando declare un miembro de una interfaz. Esta característica proporciona nuevas funcionalidades en las que puede definir implementaciones predeterminadas para las características declaradas en las interfaces. Las clases pueden elegir cuándo invalidar la funcionalidad, cuándo usar la funcionalidad predeterminada y cuándo no se debe declarar la compatibilidad con características discretas.

En este tutorial aprenderá lo siguiente:

- Creación de interfaces con implementaciones que describen características discretas.
- Creación de clases que usan las implementaciones predeterminadas.
- Creación de clases que invaliden algunas o todas las implementaciones predeterminadas.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core, incluido el compilador de C# 8.0. El compilador de C# 8.0 está disponible a partir de la versión 16.3 de Visual Studio 2019 o del SDK de .NET Core 3.0 o versiones posteriores.

Limitaciones de los métodos de extensión

Una manera de implementar el comportamiento que aparece como parte de una interfaz es definir métodos de extensión que proporcionan el comportamiento predeterminado. Las interfaces declaran un conjunto mínimo de miembros mientras proporcionan un área expuesta más grande para cualquier clase que implemente esa interfaz. Por ejemplo, los métodos de extensión de Enumerable proporcionan la implementación para que cualquier secuencia sea el origen de una consulta LINQ.

Los métodos de extensión se resuelven en tiempo de compilación, mediante el tipo declarado de la variable. Las clases que implementan la interfaz pueden proporcionar una mejor implementación para cualquier método de extensión. Las declaraciones de variables deben coincidir con el tipo de implementación para que el compilador pueda elegir esa implementación. Cuando el tipo en tiempo de compilación coincide con la interfaz, las llamadas al método se resuelven en el método de extensión. Otro problema con los métodos de extensión es que se puede tener acceso a esos métodos siempre que se pueda tener acceso a la clase que contiene los métodos de extensión. Las clases no pueden declarar si deben o no deben proporcionar características declaradas en los métodos de extensión.

A partir de C# 8.0, puede declarar las implementaciones predeterminadas como métodos de interfaz. Después, cada clase usa automáticamente la implementación predeterminada. Cualquier clase que pueda proporcionar una mejor implementación puede invalidar la definición del método de interfaz con un algoritmo mejor. En un sentido, esta técnica suena de forma similar a como se podían usar los métodos de extensión.

En este artículo, aprenderá cómo las implementaciones de interfaces predeterminadas habilitan nuevos escenarios.

Diseño de la aplicación

Considere una aplicación de automatización de dispositivos del hogar. Probablemente tenga muchos tipos

diferentes de luces e indicadores que podrían usarse en toda la casa. Cada luz debe admitir las API para encenderla y apagarla, y para notificar el estado actual. Algunas luces e indicadores pueden admitir otras características, como:

- Encender la luz y apagarla después de un tiempo.
- Hacer parpadear la luz durante un período.

Algunas de estas funcionalidades extendidas se pueden emular en los dispositivos que admiten el conjunto mínimo. Lo que indica que se proporciona una implementación predeterminada. En el caso de los dispositivos que tienen más funcionalidades integradas, el software del dispositivo usaría las funcionalidades nativas. Para otras luces, podrían optar por implementar la interfaz y usar la implementación predeterminada.

Los miembros de interfaz predeterminados son una solución mejor para este escenario que los métodos de extensión. Los autores de clases pueden controlar qué interfaces deciden implementar. Las interfaces que elijan están disponibles como métodos. Además, dado que los métodos de interfaz predeterminados son virtuales de forma predeterminada, el envío del método siempre elige la implementación en la clase.

Vamos a crear el código para mostrar estas diferencias.

Creación de interfaces

Empiece por crear la interfaz que define el comportamiento de todas las luces:

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

Un accesorio básico de luces de techo puede implementar esta interfaz, tal como se muestra en el código siguiente:

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

En este tutorial, el código no dispone de dispositivos IoT, pero emula esas actividades escribiendo mensajes en la consola. Puede explorar el código sin automatizar los dispositivos de hogar.

A continuación, vamos a definir la interfaz para una luz que se pueda apagar automáticamente después de un tiempo de espera:

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

Podría agregar una implementación básica a la luz de techo, pero una solución mejor es modificar esta definición de interfaz para proporcionar una implementación predeterminada `virtual`:

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

Al agregar ese cambio, la clase `OverheadLight` puede implementar la función de temporizador mediante la declaración de la compatibilidad con la interfaz:

```
public class OverheadLight : ITimerLight { }
```

Un tipo de luz diferente puede admitir un protocolo más sofisticado. Puede proporcionar su propia implementación de `TurnOnFor`, como se muestra en el código siguiente:

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

A diferencia de los métodos de clase virtual de invalidación, la declaración de `TurnOnFor` en la clase `HalogenLight` no utiliza la palabra clave `override`.

Funcionalidades de combinación

Las ventajas de los métodos de interfaz predeterminados resultan más claras a medida que se introducen funcionalidades más avanzadas. El uso de interfaces permite combinar las funcionalidades. También permite que cada autor de clase elija entre la implementación predeterminada y una implementación personalizada. Vamos a agregar una interfaz con una implementación predeterminada para una luz parpadeante:

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for IBlinkingLight.Blink.");
    }
}
```

La implementación predeterminada permite que cualquier luz parpadee. La luz de techo puede agregar las funcionalidades de temporizador y de parpadeo mediante la implementación predeterminada:

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

Un nuevo tipo de luz, la clase `LEDLight`, admite la función de temporizador y la función de parpadeo directamente. Este estilo de luz implementa las interfaces `ITimerLight` y `IBlinkingLight`, e invalida el método `Blink`:

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink funtion.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

La clase `ExtraFancyLight` podría admitir funciones de parpadeo y de temporizador directamente:

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

La clase `HalogenLight` que ha creado anteriormente no admite el parpadeo. Por lo tanto, no agregue la interfaz `IBlinkingLight` a la lista de interfaces compatibles.

Detección de tipos de luz mediante la coincidencia de patrones

A continuación, vamos a escribir código de prueba. Puede usar la característica de C# de coincidencia de patrones para determinar las funcionalidades de una luz mediante el examen de las interfaces que admite. El método siguiente ejercita las funcionalidades admitidas por cada luz:

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\tAfter switching on, the light is {(light.IsOn() ? "on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\tAfter switching off, the light is {(light.IsOn() ? "on" : "off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\tTimer function completed");
    }
    else
    {
        Console.WriteLine("\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\tBlink function completed");
    }
    else
    {
        Console.WriteLine("\tBlink function not supported.");
    }
}
```

El código siguiente en el método `Main` crea cada tipo de luz secuencialmente y lo prueba:

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

Cómo determina el compilador la mejor implementación

En este escenario se muestra una interfaz base sin ninguna implementación. Agregar un método a la interfaz `ILight` presenta nuevas complejidades. Las reglas del lenguaje que rigen los métodos de interfaz predeterminados minimizan el efecto en las clases concretas que implementan varias interfaces derivadas. Vamos a mejorar la interfaz original con un nuevo método para mostrar cómo cambia su uso. Cada luz del indicador puede informar de su estado de energía como un valor enumerado:

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

La implementación predeterminada presupone la ausencia de alimentación:

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

Estos cambios se compilan correctamente, aunque la clase `ExtraFancyLight` declara la compatibilidad con la interfaz de `ILight` y las interfaces derivadas `ITimerLight` y `IBlinkingLight`. Solo hay una implementación "más cercana" declarada en la interfaz `ILight`. Cualquier clase que declare una invalidación se convertirá en la implementación "más cercana". Ha visto ejemplos en las clases anteriores que reemplazaron los miembros de otras interfaces derivadas.

Evite reemplazar el mismo método en varias interfaces derivadas. Al hacerlo, se crea una llamada de método

ambiguo siempre que una clase implementa ambas interfaces derivadas. El compilador no puede elegir un solo método mejor para que emita un error. Por ejemplo, si tanto `IBlinkingLight` como `ITimerLight` implementaron una invalidación de `PowerStatus`, `OverheadLight` tendría que proporcionar una invalidación más específica. De lo contrario, el compilador no puede elegir entre las implementaciones en las dos interfaces derivadas. Normalmente, puede evitar esta situación al mantener las definiciones de interfaz pequeñas y centradas en una característica. En este escenario, cada funcionalidad de una luz es su propia interfaz; solo las clases heredan varias interfaces.

En este ejemplo se muestra un escenario en el que puede definir características discretas que se pueden combinar en clases. Declare cualquier conjunto de funcionalidades admitidas declarando qué interfaces admite una clase. El uso de métodos de interfaz predeterminados virtuales permite a las clases usar o definir una implementación diferente para cualquiera de los métodos de interfaz o para todos. Esta funcionalidad del lenguaje proporciona nuevas formas de modelar los sistemas reales que se están compilando. Los métodos de interfaz predeterminados proporcionan una forma más clara de expresar clases relacionadas que pueden combinar con diferentes características mediante implementaciones virtuales de esas funcionalidades.

Índices y rangos

19/03/2020 • 9 minutes to read • Edit Online

Los intervalos e índices proporcionan una sintaxis concisa para acceder a elementos únicos o intervalos en una secuencia.

En este tutorial aprenderá lo siguiente:

- Usar la sintaxis para intervalos de una secuencia.
- Comprender las decisiones de diseño para iniciar y finalizar cada secuencia.
- Descubrir escenarios para los tipos Index y Range.

Compatibilidad con idiomas para los índices y los rangos

Esta compatibilidad con lenguajes se basa en dos nuevos tipos y dos nuevos operadores:

- System.Index representa un índice en una secuencia.
- Índice desde el operador final `^`, que especifica que un índice es relativo al final de una secuencia.
- System.Range representa un subrango de una secuencia.
- El operador de intervalo `..`, que especifica el inicio y el final de un intervalo como sus operandos.

Comencemos con las reglas de los índices. Considere un elemento `sequence` de matriz. El índice `0` es igual que `sequence[0]`. El índice `^0` es igual que `sequence[sequence.Length]`. La expresión `sequence[^0]` produce una excepción, al igual que `sequence[sequence.Length]`. Para cualquier número `n`, el índice `^n` es igual que `sequence[sequence.Length - n]`.

```
string[] words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

Puede recuperar la última palabra con el índice `^1`. Agregue el código siguiente a la inicialización:

```
Console.WriteLine($"The last word is {words[^1]}");
```

Un rango especifica el *inicio* y el *final* de un intervalo. Los rangos son excluyentes, lo que significa que el *final* no se incluye en el intervalo. El rango `[0..^0]` representa todo el intervalo, al igual que `[0..sequence.Length]` representa todo el intervalo.

El siguiente código crea un subrango con las palabras "quick", "brown" y "fox". Va de `words[1]` a `words[3]`. El elemento `words[4]` no se encuentra en el intervalo. Agregue el código siguiente al mismo método. Cópielo y péguelo en la parte inferior de la ventana interactiva.

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write($"< {word} >");
Console.WriteLine();
```

El siguiente código crea un subrango con "lazy" y "dog". Incluye `words[^2]` y `words[^1]`. El índice del final `words[^0]` no se incluye. Agregue el código siguiente también:

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write($"< {word} >");
Console.WriteLine();
```

En los ejemplos siguientes se crean rangos con final abierto para el inicio, el final o ambos:

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the, "lazy" and "dog"
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

También puede declarar rangos o índices como variables. La variable se puede usar luego dentro de los caracteres `[` y `]`:

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

El ejemplo siguiente muestra muchos de los motivos para esas opciones. Modifique `x`, `y` y `z` para probar diferentes combinaciones. Al experimentar, use valores donde `x` sea menor que `y` y `y` sea menor que `z` para las combinaciones válidas. Agregue el código siguiente a un nuevo método. Pruebe diferentes combinaciones:

```
int[] numbers = Enumerable.Range(0, 100).ToArray();
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length - x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]}, numbers[y..z] is {y_z[0]} through
{y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with {x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as {zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..{z_zero[^1]}");
```

Compatibilidad con tipos para los índices y los rangos

Los índices y los intervalos proporcionan una sintaxis clara y concisa para acceder a un único elemento o a un subconjunto de elementos de una secuencia. Normalmente, una expresión de índice devuelve el tipo de los elementos de una secuencia. Una expresión de rango suele devolver el mismo tipo de secuencia que la secuencia de origen.

Cualquier tipo que proporcione un indexador con un parámetro Index o Range admite de manera explícita índices o rangos, respectivamente. Un indexador que toma un único parámetro Range puede devolver un tipo de secuencia diferente, como System.Span<T>.

Un tipo es **contable** si tiene una propiedad denominada `Length` o `Count` con un captador accesible y un tipo de valor devuelto de `int`. Un tipo contable que no admite índices ni rangos de manera explícita podría admitirlos implícitamente. Para más información, consulte las secciones Compatibilidad implícita de índices y Compatibilidad implícita de rangos de la nota de propuesta de características. Los rangos que usan la compatibilidad implícita del rango devuelven el mismo tipo de secuencia que la secuencia de origen.

Por ejemplo, los tipos de .NET siguientes admiten tanto índices como rangos: String, Span<T> y ReadOnlySpan<T>. List<T> admite índices, pero no rangos.

Array tiene un comportamiento con más matices. Así, las matrices de una sola dimensión admiten índices y rangos, mientras que las matrices multidimensionales no. El indexador de una matriz multidimensional tiene varios parámetros, no un parámetro único. Las matrices escalonadas, también denominadas matriz de matrices, admiten tanto intervalos como indexadores. En el siguiente ejemplo se muestra cómo iterar por una subsección rectangular de una matriz escalonada. Se itera por la sección del centro, excluyendo la primera y las últimas tres filas, así como la primera y las dos últimas columnas de cada fila seleccionada:

```
var jagged = new int[10][]
{
   new int[10] { 0,  1, 2, 3, 4, 5, 6, 7, 8, 9},
   new int[10] { 10,11,12,13,14,15,16,17,18,19},
   new int[10] { 20,21,22,23,24,25,26,27,28,29},
   new int[10] { 30,31,32,33,34,35,36,37,38,39},
   new int[10] { 40,41,42,43,44,45,46,47,48,49},
   new int[10] { 50,51,52,53,54,55,56,57,58,59},
   new int[10] { 60,61,62,63,64,65,66,67,68,69},
   new int[10] { 70,71,72,73,74,75,76,77,78,79},
   new int[10] { 80,81,82,83,84,85,86,87,88,89},
   new int[10] { 90,91,92,93,94,95,96,97,98,99},
};

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

Escenarios para los índices y los rangos

A menudo usaremos rangos e índices cuando queramos analizar un subrango de una secuencia más grande. La nueva sintaxis es más clara al leer exactamente lo que implica el subrango. La función local `MovingAverage` toma un Range como su argumento. El método enumera solo ese rango al calcular el mínimo, el máximo y la media. Pruebe con el código siguiente en su proyecto:

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax: {max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax: {max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) =>
    Enumerable.Range(0, count).Select(x => (int)(Math.Sqrt(x) * 100)).ToArray();
```

Tutorial: Expresar la intención del diseño con mayor claridad con tipos de referencia que aceptan valores NULL y que no aceptan valores NULL

02/04/2020 • 19 minutes to read • Edit Online

C# 8.0 presenta tipos de referencia que admiten un valor NULL, que complementan a los tipos de referencia del mismo modo que los tipos de valor que admiten valores NULL complementan a los tipos de valor. Declarará una variable para que sea un **tipo de referencia que acepta valores NULL** anexando un elemento `?` al tipo. Por ejemplo, `string?` representa un elemento `string` que acepta valores NULL. Puede utilizar estos nuevos tipos para expresar más claramente la intención del diseño: algunas variables *siempre deben tener un valor*, y a otras *les puede faltar un valor*.

En este tutorial aprenderá lo siguiente:

- Incorporar los tipos de referencia que aceptan valores NULL y que no aceptan valores NULL en los diseños
- Habilitar las comprobaciones de tipos de referencia que aceptan valores NULL en todo el código
- Escribir código en la parte en la que el compilador aplica esas decisiones de diseño
- Usar la característica de referencia que acepta valores NULL en sus propios diseños

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core, incluido el compilador de C# 8.0. El compilador de C# 8.0 está disponible con Visual Studio 2019 o .NET Core 3.0.

En este tutorial se da por supuesto que está familiarizado con C# y. NET, incluidos Visual Studio o la CLI de .NET Core.

Incorporación de los tipos de referencia que aceptan valores NULL en los diseños

En este tutorial, va a crear una biblioteca que modela la ejecución de una encuesta. El código usa tipos de referencia que aceptan valores NULL y tipos de referencia que no aceptan valores NULL para representar los conceptos del mundo real. Las preguntas de la encuesta nunca pueden aceptar valores NULL. Un encuestado podría preferir no responder a una pregunta. En este caso, las respuestas podrían ser `null`.

El código que escriba para este ejemplo expresa dicha intención y el compilador la exige.

Creación de la aplicación y habilitación de los tipos de referencia que aceptan valores NULL

Cree una aplicación de consola en Visual Studio o desde la línea de comandos mediante `dotnet new console`. Asigne a la aplicación el nombre `NullableIntroduction`. Una vez que se haya creado la aplicación, se deberá especificar que todo el proyecto se compila en un **contexto de anotaciones que admite un valor NULL** habilitado. Abra el archivo *.csproj* y agregue un elemento `Nullable` al elemento `PropertyGroup`. Establezca su valor en `enable`. Debe optar por recibir la característica de **tipos de referencia que admiten un valor NULL**, incluso en proyectos de C# 8.0. El motivo es que, una vez que la característica está activada, las declaraciones de variables de referencia existentes se convierten en **tipos de referencia que no aceptan valores NULL**. Aunque esa decisión lo ayudará a detectar problemas donde el código existente puede no tener comprobaciones

de valores NULL adecuadas, es posible que no se refleje con precisión la intención del diseño original:

```
<Nullable>enable</Nullable>
```

Diseño de los tipos para la aplicación

Esta aplicación de encuesta requiere la creación de una serie de clases:

- Una clase que modela la lista de preguntas
- Una clase que modela una lista de personas contactadas para la encuesta
- Una clase que modela las respuestas de una persona que realizó la encuesta

Estos tipos usarán ambas los tipos de referencia que aceptan valores NULL y los que no aceptan valores NULL para expresar qué miembros que son necesarios y cuáles opcionales. Los tipos de referencia que aceptan valores NULL comunican claramente esa intención de diseño:

- Las preguntas que forman parte de la encuesta nunca pueden ser NULL: no tiene sentido formular una pregunta vacía.
- Los encuestados nunca pueden aceptar valores NULL. Quiere realizar un seguimiento de las personas contactadas, incluso de los encuestados que han rechazado participar.
- Cualquier respuesta a una pregunta puede tener valores NULL. Los encuestados pueden negarse a responder a algunas preguntas o a todas.

Si ha programado en C#, puede estar tan acostumbrado a los tipos de referencia que permiten valores `null` que puede haber perdido otras oportunidades para declarar instancias que no admiten un valor NULL:

- La colección de preguntas no debe aceptar valores NULL.
- La colección de encuestados debe aceptar valores NULL.

A medida que escriba el código, verá que un tipo de referencia que no admite un valor NULL, como el predeterminado para las referencias, evita errores comunes que podrían generar NullReferenceException. Una lección de este tutorial es que tomó decisiones sobre qué variables podrían ser o no `null`. El lenguaje no proporcionó una sintaxis para expresar esas decisiones. Ahora sí.

La aplicación que compilará realiza los pasos siguientes:

1. Crea una encuesta y agrega preguntas a dicha encuesta.
2. Crea un conjunto de encuestados pseudoaleatorios para la encuesta.
3. Se pone en contacto con los encuestados hasta que el tamaño de la encuesta completada alcanza el número objetivo.
4. Escribe estadísticas importantes en las respuestas de la encuesta.

Compilación de la encuesta con tipos de referencia que aceptan y no aceptan valores NULL

El primer código que escriba crea la encuesta. Deberá escribir clases para modelar una pregunta de encuesta y una ejecución de encuesta. La encuesta tiene tres tipos de preguntas, que se distinguen por el formato de la respuesta: respuestas afirmativas/negativas, respuestas numéricas y respuestas de texto. Cree una clase `public SurveyQuestion`:

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

El compilador interpreta cada declaración de variable de tipo de referencia como un tipo de referencia **que no admite un valor NULL** para el código en un contexto de anotaciones que admite un valor NULL habilitado. Puede ver la primera advertencia agregando propiedades para el texto de la pregunta y el tipo de pregunta, tal y como se muestra en el código siguiente:

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

Dado que no ha inicializado `QuestionText`, el compilador emite una advertencia de que aún no se ha inicializado una propiedad que no acepta valores NULL. Su diseño requiere que el texto de la pregunta no acepte valores NULL, por lo que debe agregar un constructor para inicializar ese elemento y el valor `QuestionType` también. La definición de clase finalizada es similar al código siguiente:

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }

        public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
            (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
    }
}
```

Al agregar el constructor, se quita la advertencia. El argumento del constructor también es un tipo de referencia que no acepta valores NULL, por lo que el compilador no emite advertencias.

A continuación, cree una clase `public` denominada "`SurveyRun`". Esta clase contiene una lista de objetos `SurveyQuestion` y métodos para agregar preguntas a la encuesta, tal como se muestra en el código siguiente:

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) => surveyQuestions.Add(surveyQuestion);
    }
}
```

Al igual que antes, debe inicializar el objeto de lista en un valor distinto a NULL o el compilador emitirá una advertencia. No hay ninguna comprobación de valores que aceptan valores NULL en la segunda sobrecarga de `AddQuestion` porque no son necesarias: ha declarado esa variable para que no acepte valores NULL. Su valor no puede ser `null`.

Cambie a *Program.cs* en el editor y reemplace el contenido de `Main` con las siguientes líneas de código:

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many times (to the nearest 100) has that
happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Dado que todo el proyecto está habilitado para un contexto de anotaciones que admite un valor NULL, al pasar `null` a cualquier método que espera un tipo de referencia que no admite un valor NULL, recibirá una advertencia. Pruébelo agregando la siguiente línea a `Main`:

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Creación de los encuestados y obtención de respuestas a la encuesta

A continuación, escriba el código que genera respuestas a la encuesta. Este proceso implica realizar varias pequeñas tareas:

1. Crear un método que genere objetos de encuestados. Estos representan a las personas a las que se les ha pedido que completen la encuesta.
2. Crear una lógica para simular la realización de preguntas a un encuestado y la recopilación de respuestas o de la ausencia de respuesta de un encuestado.
3. Repetir todo esto hasta que hayan respondido a la encuesta los suficientes encuestados.

Necesitará una clase para representar una respuesta de encuesta, así que agréguela en este momento. Habilite la compatibilidad con la aceptación de valores NULL. Agregue una propiedad `Id` y un constructor que la inicialice, tal como se muestra en el código siguiente:

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

A continuación, agregue un método `static` para crear nuevos participantes mediante la generación de un identificador aleatorio:

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new SurveyResponse(randomGenerator.Next());
```

La responsabilidad principal de esta clase es generar las respuestas para que un participante de las preguntas de la encuesta. Esta responsabilidad implica una serie de pasos:

1. Solicitar la participación en la encuesta. Si la persona no da su consentimiento, devolver una respuesta con valores ausentes (o NULL).
2. Realizar cada pregunta y registrar la respuesta. Las respuestas también pueden tener valores ausentes (o NULL).

Agregue el siguiente código a la clase `SurveyResponse`:

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGHHH!";
    }
}
```

El almacenamiento de las respuestas de la encuesta es una cadena `Dictionary<int, string>?`, que indica que puede aceptar valores NULL. Está usando la nueva característica de lenguaje para declarar la intención de diseño tanto para el compilador como para cualquiera que lea el código más adelante. Si alguna vez desreferencia `surveyResponses` sin comprobar el valor `null` en primer lugar, obtendrá una advertencia del compilador. No recibirá una advertencia en el método `AnswerSurvey` porque el compilador puede determinar que la variable `surveyResponses` se estableció en un valor distinto de NULL.

Al usar `null` en las respuestas que faltan se resalta un punto clave para trabajar con tipos de referencia que aceptan valores NULL: el objetivo no es quitar todos los valores `null` del programa. En cambio, de lo que se trata es de garantizar que el código que escribe expresa la intención del diseño. Los valores que faltan son un concepto necesario para expresar en el código. El valor `null` es una forma clara de expresar los valores que faltan. El intento de quitar todos los valores `null` solo lleva a definir alguna otra forma de expresar esos valores que faltan sin `null`.

A continuación, deberá escribir el método `PerformSurvey` en la clase `SurveyRun`. Agregue el código siguiente a la clase `SurveyRun`:

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

De nuevo, la elección de que un elemento `List<SurveyResponse>?` acepte valores NULL indica que la respuesta puede tener valores NULL. Esto indica que la encuesta no se ha asignado a los encuestados todavía. Tenga en cuenta que los encuestados se agregan hasta que hayan dado su consentimiento.

El último paso para ejecutar la encuesta es agregar una llamada al realizar la encuesta al final del método `Main`:

```
surveyRun.PerformSurvey(50);
```

Examen de las respuestas de la encuesta

El último paso es mostrar los resultados de la encuesta. Agregará código a muchas de las clases que ha escrito. Este código muestra el valor de distinguir entre tipos de referencia que aceptan valores NULL y que no aceptan valores NULL. Empiece agregando los siguientes dos miembros con forma de expresión a la clase `SurveyResponse`:

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index) ?? "No answer";
```

Dado que `surveyResponses` es un tipo de referencia que admite un valor NULL, las comprobaciones de valores NULL son necesarias antes de desreferenciarlo. El método `Answer` devuelve una cadena que no admite un valor NULL, por lo que tenemos que cubrir el caso de que falte una respuesta mediante el operador de fusión de NULL.

A continuación, agregue estos tres miembros con forma de expresión a la clase `SurveyRun`:

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ?? Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

El miembro `AllParticipants` debe tener en cuenta que la variable `respondents` podría ser aceptar valores NULL, pero el valor devuelto no puede ser NULL. Si cambia esa expresión quitando `??` y la secuencia vacía de a continuación, el compilador advertirá de que el método podría devolver `null` y su firma de devolución devuelve un tipo que no acepta valores NULL.

Finalmente, agregue el siguiente bucle a la parte inferior del método `Main`:

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i).QuestionText} : {answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

No necesita ninguna comprobación `null` en este código porque ha diseñado las interfaces subyacentes para que devuelvan todos los tipos de referencia que no aceptan valores NULL.

Obtención del código

Puede obtener el código del tutorial terminado en nuestro repositorio de ejemplos en la carpeta csharp/NullableIntroduction.

Experimente cambiando las declaraciones de tipos entre tipos de referencia que aceptan valores NULL y que no aceptan valores NULL. Vea cómo así se generan advertencias diferentes para garantizar que no se desreferencia accidentalmente un valor `null`.

Pasos siguientes

Más información sobre cómo migrar una aplicación existente para usar tipos de referencia que aceptan valores NULL:

Actualización de aplicaciones para usar tipos de referencia que aceptan valores NULL

Tutorial: Migración del código existente con tipos de referencia que admiten valores NULL

18/03/2020 • 26 minutes to read • Edit Online

C#8 presenta **tipos de referencia que aceptan valores NULL**, qué complementan a los tipos de referencia del mismo modo que los tipos de valor que aceptan valores NULL complementan a los tipos de valor. Declarará una variable para que sea un **tipo de referencia que acepta valores NULL** anexando un elemento `?` al tipo. Por ejemplo, `string?` representa un elemento `string` que acepta valores NULL. Puede utilizar estos nuevos tipos para expresar más claramente la intención del diseño: algunas variables *siempre deben tener un valor*, y a otras *les puede faltar un valor*. Las variables existentes de un tipo de referencia se interpretarían como un tipo de referencia que no admite valores NULL.

En este tutorial aprenderá lo siguiente:

- Habilitar las comprobaciones de referencia NULL mientras trabaja con código.
- Diagnosticar y corregir diferentes advertencias relacionadas con los valores NULL.
- Administrar la interfaz entre contextos habilitados y deshabilitados que admiten valores NULL.
- Controlar contextos de anotación que admiten valores NULL.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core, incluido el compilador de C# 8.0. El compilador de C# 8 está disponible a partir de la versión 16.3 de Visual Studio 2019 o del SDK de .NET Core 3.0.

En este tutorial se da por supuesto que está familiarizado con C# y. NET, incluidos Visual Studio o la CLI de .NET Core.

Exploración de la aplicación de ejemplo

La aplicación de ejemplo que va a migrar es una aplicación web de lector de fuentes RSS. Esta aplicación lee una única fuente RSS y muestra los resúmenes de los artículos más recientes. Puede seleccionar cualquiera de los artículos para visitar el sitio. La aplicación es relativamente nueva, pero se escribió antes de que estuvieran disponibles los tipos de referencia que admiten valores NULL. Las decisiones de diseño para la aplicación representaban principios de sonido, pero no aprovechan esta importante característica del lenguaje.

La aplicación de ejemplo incluye una biblioteca de pruebas unitarias que valida la funcionalidad principal de la aplicación. Ese proyecto facilitará la actualización de forma segura si cambia cualquiera de las implementaciones en función de las advertencias generadas. Puede descargar el código de inicio del repositorio dotnet/samples de GitHub.

El objetivo al migrar un proyecto debe ser aprovechar las nuevas características del lenguaje para que exprese claramente su intención sobre la capacidad de admitir valores NULL de las variables, y hacerlo de tal manera que el compilador no genere advertencias cuando los contextos de anotación y advertencia que admiten valores NULL estén establecidos en `enabled`.

Actualización de proyectos a C# 8

Un buen primer paso es determinar el ámbito de la tarea de migración. Empiece por actualizar el proyecto a C# 8.0 (o posterior). Agregue el elemento `LangVersion` a PropertyGroup en ambos archivos csproj para el proyecto web y el proyecto de prueba unitaria:

```
<LangVersion>8.0</LangVersion>
```

La actualización de la versión de idioma selecciona C# 8.0, pero no habilita los contextos de anotación y de advertencia que admiten valores NULL. Recompile el proyecto para asegurarse de que se compila sin advertencias.

Un buen paso es activar el contexto de anotación que admite valores NULL y ver cuántas advertencias se generan. Agregue el siguiente elemento a ambos archivos csproj de la solución, directamente bajo el elemento `LangVersion`:

```
<Nullable>enable</Nullable>
```

Realice una compilación de prueba y observe la lista de advertencias. En esta pequeña aplicación, el compilador genera cinco advertencias, por lo que es probable que dejara el contexto de anotación que admite valores NULL habilitado y empezara a corregir las advertencias para todo el proyecto.

Esa estrategia solo funciona para los proyectos más pequeños. Para los proyectos más grandes, el número de advertencias generadas al habilitar el contexto de anotación que admite valores NULL para todo el código base dificulta la corrección de las advertencias de forma sistemática. Para los proyectos empresariales más grandes, suele ser conveniente migrar los proyectos de uno en uno. En cada proyecto, migre una clase o un archivo de uno en uno.

Las advertencias ayudan a detectar la intención del diseño original

Hay dos clases que generan varias advertencias. Empiece con la clase `NewsStoryViewModel`. Quite el elemento `Nullable` de ambos archivos csproj de forma que pueda limitar el ámbito de las advertencias para las secciones de código con las que está trabajando. Abra el archivo *NewsStoryViewModel.cs* y agregue las siguientes directivas para habilitar el contexto de anotación que admite valores NULL para `NewsStoryViewModel` y restáurelo siguiendo esa definición de clase:

```
#nullable enable
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
#nullable restore
```

Estas dos directivas le ayudan a centrar sus esfuerzos en la migración. Las advertencias que admiten valores NULL se generan para el área del código en el que está trabajando activamente. Podrá dejarlas hasta que esté listo para activar las advertencias para todo el proyecto. Debe usar `restore` en lugar del valor `disable` para que no deshabilite accidentalmente el contexto más adelante cuando haya activado las anotaciones que admiten valores NULL para todo el proyecto. Cuando haya activado el contexto de anotación que admite valores NULL para todo el proyecto, podrá quitar todas las pragmas `#nullable` de ese proyecto.

La clase `NewsStoryViewModel` es un objeto de transferencia de datos (DTO) y dos de las propiedades son cadenas de lectura y escritura:

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
```

Estas dos propiedades generan `CS8618`, "La propiedad que admite valores NULL no está inicializada". Eso es muy claro: ambas propiedades `string` tienen el valor predeterminado de `null` cuando `NewsStoryViewModel` se construye. Lo que es importante descubrir es cómo se construyen los objetos `NewsStoryViewModel`. Examinando esta clase no puede saber si el valor de `null` es parte del diseño o si estos objetos se establecen en valores que no admiten valores NULL siembre que se crea uno. Los artículos de noticias se crean en el método `GetNews` de la clase `NewsService`:

```
ISyndicationItem item = await feedReader.ReadItem();
var newsStory = _mapper.Map<NewsStoryViewModel>(item);
news.Add(newsStory);
```

Suceden algunas cosas en el bloque de código anterior. Esta aplicación usa el paquete NuGet AutoMapper para construir un elemento de noticias a partir de `ISyndicationItem`. Ha detectado que los elementos de artículo de noticias se construyen y las propiedades se establecen en esa instrucción. Esto significa que el diseño de `NewsStoryViewModel` indica que estas propiedades nunca deben tener el valor `null`. Estas propiedades deben ser **tipos de referencia que no admiten valores NULL**. Eso expresa mejor la intención del diseño original. De hecho, para `NewsStoryViewModel`, *se* crean instancias correctamente con valores no NULL. Eso convierte al siguiente código de inicialización en una corrección válida:

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; } = default!;
    public string Uri { get; set; } = default!;
}
```

La asignación de `Title` y `Uri` a `default` que es `null` para el `string` tipo no cambia el comportamiento en tiempo de ejecución del programa. `NewsStoryViewModel` se sigue construyendo con valores NULL, pero ahora el compilador no informa de ninguna advertencia. El **operador que permite valores NULL**, el carácter `!` que sigue a la expresión `default`, indica al compilador que la expresión anterior no es NULL. Esta técnica puede ser conveniente cuando otros cambios fuerzan cambios mucho mayores en una base de código, pero en esta aplicación hay una solución relativamente rápida y mejor: convierta a `NewsStoryViewModel` en un tipo inmutable donde todas las propiedades se establecen en el constructor. Realice los cambios siguientes en `NewsStoryViewModel`:

```
#nullable enable
    public class NewsStoryViewModel
    {
        public NewsStoryViewModel(DateTimeOffset published, string title, string uri) =>
            (Published, Title, Uri) = (published, title, uri);

        public DateTimeOffset Published { get; }
        public string Title { get; }
        public string Uri { get; }
    }
#nullable restore
```

Una vez hecho esto, deberá actualizar el código que configura AutoMapper para que utilice el constructor en lugar

de establecer las propiedades. Abra `NewsService.cs` y busque el código siguiente en la parte inferior del archivo:

```
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForMember(dest => dest.Uri, opts => opts.MapFrom(src => src.Id));
    }
}
```

Ese código asigna las propiedades del objeto `ISyndicationItem` a las propiedades de `NewsStoryViewModel`. Desea AutoMapper para proporcionar la asignación mediante un constructor. Reemplace el código anterior con la siguiente configuración de AutoMapper:

```
#nullable enable
    public class NewsStoryProfile : Profile
    {
        public NewsStoryProfile()
        {
            // Create the AutoMapper mapping profile between the 2 objects.
            // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
            CreateMap<ISyndicationItem, NewsStoryViewModel>()
                .ForCtorParam("uri", opt => opt.MapFrom(src => src.Id));
        }
```

Tenga en cuenta que, dado que esta clase es pequeña y ha realizado un examinen meticuloso, debe activar la directiva `#nullable enable` sobre esta declaración de clase. El cambio en el constructor podría haberse interrumpido en cierta medida, por lo que vale la pena ejecutar todas las pruebas y probar la aplicación antes de continuar.

El primer conjunto de cambios le mostraba cómo detectar cuándo el diseño original indicaba que las variables no debían establecerse en `null`. La técnica se conoce como **correcto por construcción**. Declara que un objeto y sus propiedades no pueden ser `null` cuando se construye. El análisis de flujo del compilador proporciona garantía de que esas propiedades no están establecidas en `null` después de la construcción. Tenga en cuenta que el código externo llama a este constructor y, dicho código, es **ajeno a la admisión de valores NULL**. La nueva sintaxis no proporciona comprobación en tiempo de ejecución. El código externo podría eludir el análisis de flujo del compilador.

En otras ocasiones, la estructura de una clase proporciona pistas diferentes a la intención. Abra el archivo *Error.cshtml.cs* en la carpeta *Páginas*. `ErrorViewModel` contiene el código siguiente:

```
public class ErrorModel : PageModel
{
    public string RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}
```

Agregue la directiva `#nullable enable` antes de la declaración de clase y una directiva `#nullable restore` después de ella. Recibirá una advertencia que indica que `RequestId` no se ha inicializado. Si examina la clase, debe decidir

que la propiedad `RequestId` debe ser NULL en algunos casos. La existencia de la propiedad `ShowRequestId` indica que puede haber valores que faltan. Dado que `null` es válido, agregue `?` en el tipo `string` para indicar que la propiedad `RequestId` es un *tipo de referencia que admite valores NULL*. La clase final se parece al ejemplo siguiente:

```
#nullable enable
    public class ErrorModel : PageModel
    {
        public string? RequestId { get; set; }

        public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

        public void OnGet()
        {
            RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
        }
    }
#nullable restore
```

Compruebe los usos de la propiedad y verá que, en la página asociada, se comprueba si la propiedad es NULL antes de representarla en el marcado. Ese es un uso seguro de un tipo de referencia que admite valores NULL, por lo que ha terminado con esta clase.

La corrección de valores NULL provoca cambio

Con frecuencia, la corrección de un conjunto de advertencias crea nuevas advertencias de código relacionado. Vamos a ver las advertencias en acción mediante la corrección de la clase `index.cshtml.cs`. Abra el archivo `index.cshtml.cs` y examine el código. Este archivo contiene el código de la página de índice:

```
public class IndexModel : PageModel
{
    private readonly NewsService _newsService;

    public IndexModel(NewsService newsService)
    {
        _newsService = newsService;
    }

    public string ErrorText { get; private set; }

    public List<NewsStoryViewModel> NewsItems { get; private set; }

    public async Task OnGet()
    {
        string feedUrl = Request.Query["feedurl"];

        if (!string.IsNullOrEmpty(feedUrl))
        {
            try
            {
                NewsItems = await _newsService.GetNews(feedUrl);
            }
            catch (UriFormatException)
            {
                ErrorText = "There was a problem parsing the URL.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.NameResolutionFailure)
            {
                ErrorText = "Unknown host name.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.ProtocolError)
            {
                ErrorText = "Syndication feed not found.";
                return;
            }
            catch (AggregateException ae)
            {
                ae.Handle((x) =>
                {
                    if (x is XmlException)
                    {
                        ErrorText = "There was a problem parsing the feed. Are you sure that URL is a
syndication feed?";
                        return true;
                    }
                    return false;
                });
            }
        }
    }
}
```

Agregue la directiva `#nullable enable` y verá dos advertencias. Ni la propiedad `ErrorText` ni la propiedad `NewsItems` se inicializan. Al examinar esta clase se llegará a la conclusión de que ambas propiedades deben ser tipos de referencia que admiten valores NULL: Ambas tienen establecedores privados. Se asigna exactamente una en el método `OnGet`. Antes de realizar cambios, examine los consumidores de ambas propiedades. En la propia página, `ErrorText` se comprueba con NULL antes de generar el marcado para los errores. La colección `NewsItems` se compara con `null` y se comprueba para asegurar que tiene elementos. Una corrección rápida sería convertir ambas propiedades en tipos de referencia que admiten valores NULL. Una corrección mejor sería convertir a la colección en un tipo de referencia que no admite valores NULL y agregar elementos a la colección existente

cuando se recuperan noticias. La primera corrección consiste en agregar `?` al tipo `string` para `ErrorText`:

```
public string? ErrorText { get; private set; }
```

Ese cambio no se propagará por otro código, porque cualquier acceso a la propiedad `ErrorText` ya estaba protegido por comprobaciones NULL. A continuación, inicialice la lista `NewsItems` y quite el establecedor de propiedad, convirtiéndola en una propiedad de solo lectura:

```
public List<NewsStoryViewModel> NewsItems { get; } = new List<NewsStoryViewModel>();
```

Esa acción corrigió la advertencia pero introdujo un error. La lista `NewsItems` es ahora **correcta por construcción**, pero el código que establece la lista en `OnGet` debe cambiar para que coincida con la nueva API. En lugar de una asignación, llame a `AddRange` para agregar los elementos de noticias a la lista existente:

```
NewsItems.AddRange(await _newsService.GetNews(feedUrl));
```

El uso de `AddRange` en lugar de una asignación significa que el método `GetNews` puede devolver `IEnumerable` en lugar de `List`. Eso guarda una asignación. Cambie la firma del método y quite la llamada `ToList`, como se muestra en el ejemplo de código siguiente:

```
public async Task<IEnumerable<NewsStoryViewModel>> GetNews(string feedUrl)
{
    var news = new List<NewsStoryViewModel>();
    var feedUri = new Uri(feedUrl);

    using (var xmlReader = XmlReader.Create(feedUri.ToString(),
           new XmlReaderSettings { Async = true }))
    {
        try
        {
            var feedReader = new RssFeedReader(xmlReader);

            while (await feedReader.Read())
            {
                switch (feedReader.ElementType)
                {
                    // RSS Item
                    case SyndicationElementType.Item:
                        ISyndicationItem item = await feedReader.ReadItem();
                        var newsStory = _mapper.Map<NewsStoryViewModel>(item);
                        news.Add(newsStory);
                        break;

                    // Something else
                    default:
                        break;
                }
            }
        }
        catch (AggregateException ae)
        {
            throw ae.Flatten();
        }
    }

    return news.OrderByDescending(story => story.Published);
}
```

El cambio de firma también rompe una de las pruebas. Abra el archivo `NewsServiceTests.cs` de la carpeta

`Services` del proyecto `SimpleFeedReader.Tests`. Vaya a la prueba `Returns_News_Stories_Given_Valid_Uri` y cambie el tipo de la variable `result` a `IEnumerable<NewsItem>`. El cambio de tipo significa que la propiedad `Count` ya no está disponible, por tanto, reemplace la propiedad `Count` en `Assert` con una llamada a `Any()`:

```
// Act
IEnumerable<NewsStoryViewModel> result =
    await _newsService.GetNews(feedUrl);

// Assert
Assert.True(result.Any());
```

También deberá agregar una instrucción `using System.Linq` al principio del archivo.

Este conjunto de cambios resalta una consideración especial al actualizar el código que incluye creación de instancias genéricas. La lista y los elementos de la lista de tipos que no admiten valores NULL. Uno o ambos podrían ser tipos que admiten valores NULL. Se permiten todas las declaraciones siguientes:

- `List<NewsStoryViewModel>`: lista que no admite valores NULL de modelos de vista que no admiten valores NULL.
- `List<NewsStoryViewModel?>`: lista que no admite valores NULL de modelos de vista que admiten valores NULL.
- `List<NewsStoryViewModel>?`: lista que admite valores NULL de modelos de vista que no admiten valores NULL.
- `List<NewsStoryViewModel?>?`: lista que admite valores NULL de modelos de vista que admiten valores NULL.

Interfaces con código externo

Ha realizado cambios en la clase `NewsService`, así que active la anotación `#nullable enable` para esa clase. Esto no generará nuevas advertencias. Sin embargo, un examen cuidadoso de la clase ayuda a ilustrar algunas de las limitaciones del análisis de flujo del compilador. Examine el constructor:

```
public NewsService(IMapper mapper)
{
    _mapper = mapper;
}
```

El parámetro `IMapper` se escribe como una referencia que no admite valores NULL. El código de infraestructura de ASP.NET Core lo llama, por lo que el compilador realmente no sabe que `IMapper` nunca será NULL. El contenedor predeterminado de inyección de dependencias (DI) de ASP.NET Core inicia una excepción si no se puede resolver un servicio necesario, por lo que el código es correcto. El compilador no puede validar todas las llamadas a las API públicas, incluso si el código se compila con contextos de anotación que acepta valores NULL habilitados. Además, las bibliotecas pueden ser consumidas por proyectos que aún no han participado mediante tipos de referencia que admiten valores NULL. Valide las entradas en API públicas aunque las haya declarado como tipos que no admiten valores NULL.

Obtención del código

Ha corregido las advertencias que identificó en la compilación de prueba inicial, así que ahora puede activar el contexto de anotación que acepta valores NULL para ambos proyectos. Vuelva a compilar los proyectos; el compilador no notifica advertencias. Puede obtener el código del proyecto finalizado en el repositorio dotnet/samples de GitHub.

Las nuevas características compatibles con tipos de referencia que admiten valores NULL le ayudan a encontrar y corregir los posibles errores en la forma de controlar valores `null` en el código. La habilitación del contexto de anotación que acepta valores NULL permite expresar la intención del diseño: algunas variables nunca deberían ser NULL y otras pueden contener valores NULL. Estas características hacen que sea más fácil declarar la intención del

diseño. De forma similar, el contexto de advertencia que admite valores NULL indica al compilador que emita advertencias cuando se haya infringido dicha intención. Esas advertencias le guían para realizar actualizaciones que hacen que el código sea más resistente y tenga menos probabilidad de iniciar `NullReferenceException` durante la ejecución. Puede controlar el ámbito de estos contextos para que pueda centrarse en áreas locales del código que desea migrar mientras el código base restante permanece intacto. En la práctica, puede hacer que esta tarea de migración sea una parte del mantenimiento convencional de las clases. En este tutorial se ha mostrado el proceso para migrar una aplicación para usar tipos de referencia que admiten valores NULL. Puede explorar un ejemplo más grande real de este proceso mediante el examen del repositorio popular Jon Skeet creado para incorporar tipos de referencia que admitan valores NULL en NodaTime. Además, puede aprender técnicas para usar tipos de referencias que aceptan valores NULL con Entity Framework Core en Entity Framework Core: trabajar con tipos de referencia que aceptan valores NULL.

Tutorial: Generación y uso de secuencias asincrónicas con C# 8.0 y .NET Core 3.0

29/06/2020 • 17 minutes to read • Edit Online

C# 8.0 presenta **secuencias asincrónicas**, que modelan un origen de datos de streaming. Las secuencias de datos suelen recuperar o generar elementos de forma asincrónica. Las secuencias asincrónicas se basan en nuevas interfaces introducidas en .NET Standard 2.1. Dichas interfaces se admiten en .NET Core 3.0 y versiones posteriores. Proporcionan un modelo de programación natural para los orígenes de datos de streaming asincrónicos.

En este tutorial aprenderá lo siguiente:

- Crear un origen de datos que genera una secuencia de elementos de datos de forma asincrónica.
- Utilizar ese origen de datos de forma asincrónica.
- Admitir la cancelación y los contextos capturados para secuencias asincrónicas.
- Reconocer cuándo la interfaz y el origen de datos nuevos son preferibles a las secuencias de datos sincrónicas anteriores.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core, incluido el compilador de C# 8.0. El compilador de C# 8 está disponible a partir de la versión 16.3 de Visual Studio 2019 o del SDK de .NET Core 3.0.

Deberá crear un token de acceso de GitHub para poder tener acceso al punto de conexión de GraphQL de GitHub. Seleccione los siguientes permisos para el token de acceso de GitHub:

- repo:status
- public_repo

Guarde el token de acceso en un lugar seguro para usarlo a fin de obtener acceso al punto de conexión de API de GitHub.

> **WARNING**
> Mantenga seguro su token de acceso personal. Cualquier software con su token de acceso personal podría realizar llamadas de API de GitHub con sus derechos de acceso.

En este tutorial se da por supuesto que está familiarizado con C# y. NET, incluidos Visual Studio o la CLI de .NET Core.

Ejecución de la aplicación de inicio

Puede obtener el código para la aplicación de inicio usada en este tutorial en el repositorio dotnet/docs de la carpeta csharp/tutoriales/AsyncStreams.

La aplicación de inicio es una aplicación de consola que usa la interfaz GraphQL de GitHub para recuperar las incidencias recientes escritas en el repositorio dotnet/docs. Comience por mirar el código siguiente para el método `Main` de la aplicación de inicio:

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-
token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment variable",
    "");

    var client = new GitHubClient(new Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new CancellationTokenSource();

    try
    {
        var results = await runPagedQueryAsync(client, PagedIssueQuery, "docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

Puede establecer una variable de entorno `GitHubKey` para el token de acceso personal, o bien puede reemplazar el último argumento en la llamada a `GenEnvVariable` por el token de acceso personal. No coloque el código de acceso en el código fuente si va a compartir el origen con otros usuarios. No cargue nunca códigos de acceso en un repositorio de código fuente compartido.

Después de crear el cliente de GitHub, el código de `Main` crea un objeto de informe de progreso y un token de cancelación. Una vez que se crean esos objetos, `Main` llama a `runPagedQueryAsync` para recuperar las 250 incidencias creadas más recientemente. Una vez finalizada esa tarea, se muestran los resultados.

Al ejecutar la aplicación inicial, puede realizar algunas observaciones importantes acerca de cómo se ejecuta esta aplicación. Verá el progreso notificado para cada página devuelta desde GitHub. Puede observar una pausa marcada antes de que GitHub devuelva cada nueva página de incidencias. Por último, se muestran las incidencias solo después de que se hayan recuperado 10 páginas de GitHub.

Examen de la implementación

La implementación revela por qué observó el comportamiento descrito en la sección anterior. Examine el código de `runPagedQueryAsync`:

```
private static async Task<JArray> runPagedQueryAsync(GitHubClient client, string queryText, string repoName,
CancellationToken cancel, IProgress<int> progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();
        finalResults.Merge(issues(results)["nodes"]);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

Vamos a concentrarnos en el algoritmo de paginación y la estructura asincrónica del código anterior. (Puede consultar la documentación de GraphQL de GitHub para obtener más información sobre la API de GraphQL de GitHub). El método `runPagedQueryAsync` enumera las incidencias desde la más reciente hasta la más antigua. Solicita 25 incidencias por página y examina la estructura `pageInfo` de la respuesta para continuar con la página anterior. Eso sigue al soporte de paginación estándar de GraphQL para respuestas de varias páginas. La respuesta incluye un objeto `pageInfo` que incluye a su vez un valor `hasPreviousPages` y un valor `startCursor` usado para solicitar la página anterior. Las incidencias se encuentran en la matriz `nodes`. El método `runPagedQueryAsync` anexa estos nodos a una matriz que contiene todos los resultados de todas las páginas.

Después de recuperar y restaurar una página de resultados, `runPagedQueryAsync` informa del progreso y comprueba la cancelación. Si se ha solicitado la cancelación, `runPagedQueryAsync` lanza un OperationCanceledException.

Hay varios elementos en este código que se pueden mejorar. Lo más importante, `runPagedQueryAsync` debe asignar el almacenamiento para todas las incidencias devueltas. Este ejemplo se detiene en 250 incidencias porque la recuperación de todas las incidencias abiertas requeriría mucha más memoria para almacenar todas las incidencias recuperadas. Los protocolos para admitir los informes de progreso y la cancelación hacen que el algoritmo sea más difícil de comprender en su primera lectura. Hay más tipos y API implicados. Debe realizar un seguimiento de las comunicaciones a través de CancellationTokenSource y su CancellationToken asociado para comprender dónde se solicita la cancelación y dónde se concede.

Las secuencias asincrónicas ofrecen una manera mejor

Las secuencias asincrónicas y el lenguaje asociado abordan todas estas cuestiones. El código que genera la secuencia ahora puede usar `yield return` para devolver los elementos en un método que se declaró con el modificador `async`. Puede usar una secuencia asincrónica utilizando un bucle `await foreach` igual que puede usar una secuencia mediante un bucle `foreach`.

Estas nuevas características del lenguaje dependen de tres nuevas interfaces agregadas a .NET Standard 2.1 e implementadas en .NET Core 3.0:

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

Estas tres interfaces deben resultar familiares a la mayoría de desarrolladores de C#. Se comportan de manera similar con sus contrapartes sincrónicas:

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

Un tipo que podría ser desconocido es System.Threading.Tasks.ValueTask. La estructura `ValueTask` proporciona una API similar a la clase System.Threading.Tasks.Task. `ValueTask` se usa en estas interfaces por motivos de rendimiento.

Conversión en secuencias asincrónicas

A continuación, convierta el método `runPagedQueryAsync` para generar una secuencia asincrónica. En primer lugar, cambie la signatura de `runPagedQueryAsync` para que devuelva un `IAsyncEnumerable<JToken>` y quite el token de cancelación y los objetos de progreso de la lista de parámetros como se muestra en el código siguiente:

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

El código de inicio procesa cada página a medida que se recupera, tal como se muestra en el código siguiente:

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Reemplace esas tres líneas por el código siguiente:

```
foreach (JObject issue in issues(results)["nodes"])
    yield return issue;
```

También puede quitar la declaración de `finalResults` anteriormente en este método y la instrucción `return` que sigue al bucle modificado.

Ha terminado los cambios para generar una secuencia asincrónica. El método finalizado debería ser similar al código siguiente:

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

A continuación, cambie el código que utiliza la colección para usar la secuencia asincrónica. Busque el código siguiente en `Main` que procesa la colección de incidencias:

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await runPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}
```

Reemplace el código por el siguiente bucle `await foreach`:

```
int num = 0;
await foreach (var issue in runPagedQueryAsync(client, PagedIssueQuery, "docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

La nueva interfaz IAsyncEnumerator<T> deriva de IAsyncDisposable. Esto significa que el bucle anterior desechará la secuencia de forma asincrónica cuando finalice el bucle. Como imaginará, el bucle es similar al código siguiente:

```
int num = 0;
var enumerator = runPagedQueryAsync(client, PagedIssueQuery, "docs").GetEnumeratorAsync();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

Los elementos de secuencia se procesan de forma predeterminada en el contexto capturado. Si quiere deshabilitar la captura del contexto, use el método de extensión TaskAsyncEnumerableExtensions.ConfigureAwait. Para obtener más información sobre los contextos de sincronización y la captura del contexto actual, vea el artículo sobre el consumo del patrón asincrónico basado en tareas.

Las secuencias asincrónicas admiten la cancelación mediante el mismo protocolo que otros métodos `async`. Para admitir la cancelación, debe modificar la firma del método de iterador asincrónico como se indica a continuación:

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation] CancellationToken cancellationToken = default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results = JObject.Parse(response.HttpResponse.Body.ToString());

        int totalCount = (int)issues(results)["totalCount"];
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"];
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)["startCursor"].ToString();
        issuesReturned += issues(results)["nodes"].Count();

        foreach (JObject issue in issues(results)["nodes"])
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]["repository"]["issues"];
    JObject pageInfo(JObject result) => (JObject)issues(result)["pageInfo"];
}
```

El atributo EnumeratorCancellationAttribute hace que el compilador genere código para IAsyncEnumerator<T>, que hace que el token que se pasa a `GetAsyncEnumerator` sea visible al cuerpo del iterador asincrónico como ese argumento. En `runQueryAsync`, puede examinar el estado del token y cancelar el trabajo posterior si es necesario.

Se puede usar otro método de extensión, WithCancellation, para pasar el token de cancelación a la secuencia asincrónica. Modifique el bucle que enumera los problemas de la siguiente manera:

```
private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in runPagedQueryAsync(client, PagedIssueQuery, "docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

Puede obtener el código para el tutorial finalizado en el repositorio dotnet/docs de la carpeta csharp/tutoriales/AsyncStreams.

Ejecución de la aplicación finalizada

Vuelva a ejecutar la aplicación. Compare su comportamiento con el comportamiento de la aplicación de inicio. La primera página de resultados se enumera en cuanto está disponible. Hay una pausa marcada cada vez que se solicita y se recupera una página nueva; a continuación, se enumeran rápidamente los resultados de la página

siguiente. El bloque `try` / `catch` no es necesario para controlar la cancelación: el autor de la llamada puede detener la enumeración de la colección. El progreso se notifica claramente porque la secuencia asincrónica genera resultados a medida que se descarga cada página. El estado de cada problema devuelto se incluye sin problemas en el bucle `await foreach`. No necesita un objeto de devolución de llamada para hacer un seguimiento del progreso.

Puede ver mejoras en el uso de la memoria examinando el código. Ya no tiene que asignar una colección para almacenar todos los resultados antes de que se enumeren. El autor de la llamada puede determinar cómo consumir los resultados y si se necesita una colección de almacenamiento.

Ejecute las aplicaciones de inicio y finalizada y podrá ver las diferencias entre las implementaciones personalmente. Puede eliminar el token de acceso de GitHub que creó cuando inició este tutorial cuando haya terminado. Si un atacante obtuviera acceso a dicho token, podría tener acceso a sus API de GitHub con sus credenciales.

Tutorial: Uso de las características de coincidencia de patrones para ampliar los tipos de datos

18/03/2020 • 26 minutes to read • Edit Online

C# 7 introdujo las características básicas de coincidencia de patrones. Esas características se amplían en C# 8 con nuevas expresiones y patrones. Puede escribir una funcionalidad que se comporta como si se hubiesen ampliado tipos que pueden estar en otras bibliotecas. Los patrones también se usan para crear una funcionalidad que la aplicación requiere y que no es una característica fundamental del tipo que se está ampliando.

En este tutorial aprenderá lo siguiente:

- Reconocer situaciones donde se debe usar la coincidencia de patrones.
- Usar las expresiones de coincidencia de patrones para implementar un comportamiento en función de los tipos y los valores de propiedad.
- Combinar la coincidencia de patrones con otras técnicas para crear algoritmos completos.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core, incluido el compilador de C# 8.0. El compilador de C# 8 está disponible a partir de la versión 16.3 de Visual Studio 2019 o del SDK de .NET Core 3.0.

En este tutorial se da por supuesto que está familiarizado con C# y. NET, incluidos Visual Studio o la CLI de .NET Core.

Escenarios para la coincidencia de patrones

Con frecuencia, el desarrollo moderno incluye integrar datos desde varios orígenes y presentar información y perspectivas a partir de esos datos en una sola aplicación cohesiva. Ni usted ni su equipo tendrán control ni acceso para todos los tipos que representan los datos entrantes.

El diseño clásico orientado a objetos llamaría a la creación de tipos en la aplicación que representen cada tipo de datos de esos orígenes de datos múltiples. Luego, la aplicación podría trabajar con esos tipos nuevos, crear jerarquías de herencia, crear métodos virtuales e implementar abstracciones. Esas técnicas funcionan y son a veces las mejores herramientas. En otras ocasiones, puede escribir menos código. Puede escribir código más claro con técnicas que separan los datos de las operaciones que manipulan esos datos.

En este tutorial, creará y explorará una aplicación que toma datos entrantes de varios orígenes externos para un solo escenario. Verá cómo la **coincidencia de patrones** proporciona una forma eficaz de consumir y procesar esos datos de maneras que no formaban parte del sistema original.

Considere un área metropolitana importante que usa peajes y precios estipulados para las horas de mayor actividad con el fin de administrar el tráfico. Puede escribir una aplicación que calcule los peajes de un vehículo en función de su tipo. Mejoras posteriores incorporan precios basados en la cantidad de ocupantes del vehículo. Otras mejoras agregan precios según la hora y el día de la semana.

Desde esa descripción breve, puede haber esbozado rápidamente una jerarquía de objetos para modelar este sistema. Sin embargo, los datos provienen de varios orígenes, como otros sistemas de administración de registros de vehículos. Estos sistemas ofrecen distintas clases para modelar esos datos y no se tiene un modelo de objetos único que puede usar. En este tutorial, usará estas clases simplificadas para modelar los datos del vehículo desde dichos sistemas externos, tal como se muestra en el código siguiente:

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

Puede descargar el código de inicio del repositorio dotnet/samples de GitHub. Puede ver que las clases de vehículo provienen de distintos sistemas y que están en distintos espacios de nombres. No se puede aprovechar ninguna clase base común distinta de `System.Object`.

Diseños de coincidencia de patrones

En el escenario que se usa en este tutorial se resaltan los tipos de problemas en los que resulta adecuado usar la coincidencia de patrones para resolverlos:

- Los objetos con los que necesita trabajar no están en una jerarquía de objetos que coincida con sus objetivos. Es posible que trabaje con clases que forman parte de sistemas no relacionados.
- La funcionalidad que agrega no forma parte de la abstracción central de estas clases. El peaje que paga un vehículo *cambia* según los distintos tipos de vehículos, pero el peaje no es una función central del vehículo.

Cuando la *forma* de los datos y las *operaciones* que se realizan en esos datos no se describen en conjunto, las características de coincidencia de patrones de C# permiten que sea más fácil trabajar con ellos.

Implementación de cálculos de peajes básicos

El cálculo de peaje más básico solo se basa en el tipo de vehículo:

- Un `Car` es USD 2,00.
- Un `Taxi` es USD 3,50.
- Un `Bus` es USD 5,00.
- Un `DeliveryTruck` es USD 10,00

Cree una clase `TollCalculator` nueva e implemente la coincidencia de patrones en el tipo de vehículo para obtener el importe del peaje. En el siguiente código se muestra la implementación inicial de `TollCalculator`.

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    public class TollCalculator
    {
        public decimal CalculateToll(object vehicle) =>
            vehicle switch
        {
            Car c           => 2.00m,
            Taxi t          => 3.50m,
            Bus b           => 5.00m,
            DeliveryTruck t => 10.00m,
            { }             => throw new ArgumentException(message: "Not a known vehicle type", paramName:
nameof(vehicle)),
            null            => throw new ArgumentNullException(nameof(vehicle))
        };
    }
}
```

El código anterior usa una **expresión switch** (que no es la misma que en una instrucción `switch`) que prueba el **patrón de tipo**. Una **expresión switch** comienza por la variable, `vehicle` en el código anterior, seguida de la palabra clave `switch`. A continuación, todos los **segmentos modificadores** aparecen entre llaves. La expresión `switch` lleva a cabo otras mejoras en la sintaxis que rodea la instrucción `switch`. La palabra clave `case` se omite y el resultado de cada segmento es una expresión. Los dos últimos segmentos muestran una característica de lenguaje nueva. El caso `{ }` coincide con cualquier objeto no nulo que no coincidía con ningún segmento anterior. Este segmento detecta todo tipo incorrecto que se pasa a este método. El caso `{ }` debe seguir los casos de cada tipo de vehículo. Si se invierte el orden, el caso `{ }` tendrá prioridad. Por último, el patrón `null` detecta si se pasa `null` a este método. El patrón `null` puede ser el último porque los otros patrones de tipos solo coinciden con un objeto no nulo del tipo correcto.

Puede probar este código con el código siguiente en `Program.cs`:

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            var tollCalc = new TollCalculator();

            var car = new Car();
            var taxi = new Taxi();
            var bus = new Bus();
            var truck = new DeliveryTruck();

            Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
            Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
            Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
            Console.WriteLine($"The toll for a truck is {tollCalc.CalculateToll(truck)}");

            try
            {
                tollCalc.CalculateToll("this will fail");
            }
            catch (ArgumentException e)
            {
                Console.WriteLine("Caught an argument exception when using the wrong type");
            }
            try
            {
                tollCalc.CalculateToll(null);
            }
            catch (ArgumentNullException e)
            {
                Console.WriteLine("Caught an argument exception when using null");
            }
        }
    }
}
```

Ese código se incluye en el proyecto de inicio, pero se marca como comentario. Quite los comentarios y podrá probar lo que escribió.

Empezará a ver cómo los patrones pueden ayudarlo a crear algoritmos cuando el código y los datos están separados. La expresión `switch` prueba el tipo y genera valores distintos en función de los resultados. Eso es solo el principio.

Incorporación de precios por ocupación

La autoridad encargada de los peajes quiere incentivar que los vehículos viajen a plena capacidad. Se decidió cobrar más cuando los vehículos circulan con menos pasajeros y se incentiva que los vehículos vayan llenos al ofrecer precios más bajos:

- Los automóviles y taxis sin pasajeros pagan USD 0,50 adicionales.
- Los automóviles y taxis con dos pasajeros tienen un descuento de USD 0,50.
- Los automóviles y taxis con tres o más pasajeros tienen un descuento de USD 1.
- Los buses que viajan con menos del 50 % de su capacidad pagan USD 2 adicionales.
- Los buses que viajan con más del 90 % de su capacidad tienen un descuento de USD 1.

Estas reglas se pueden implementar con el **patrón de propiedad** en la misma expresión switch. El patrón de propiedad examina las propiedades del objeto una vez que se determina el tipo. El caso único de `Car` se amplía a cuatro casos distintos:

```
vehicle switch
{
    Car { Passengers: 0}        => 2.00m + 0.50m,
    Car { Passengers: 1 }       => 2.0m,
    Car { Passengers: 2}        => 2.0m - 0.50m,
    Car c                       => 2.00m - 1.0m,

    // ...
};
```

Los primeros tres casos prueban el tipo como `Car` y luego comprueban el valor de la propiedad `Passengers`. Si ambos coinciden, esa expresión se evalúa y devuelve.

También podría expandir los casos para los taxis de manera similar:

```
vehicle switch
{
    // ...

    Taxi { Fares: 0}  => 3.50m + 1.00m,
    Taxi { Fares: 1 } => 3.50m,
    Taxi { Fares: 2}  => 3.50m - 0.50m,
    Taxi t            => 3.50m - 1.00m,

    // ...
};
```

En el ejemplo anterior, la cláusula `when` se omitía en el caso final.

A continuación, implemente las reglas de ocupación mediante la expansión de los casos para los buses, tal como se muestra en el ejemplo siguiente:

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus b => 5.00m,

    // ...
};
```

A la autoridad encargada de los peajes no le preocupa el número de pasajeros en los camiones de reparto. Alternativamente, ajustan el importe del peaje en base a la clase de peso de los camiones, como sigue:

- A los camiones de más de 2268 kilos se les cobran USD 5 adicionales.
- Los camiones livianos, por debajo de los 1360 kilos, tienen un descuento de 2 USD.

Esta regla se implementa con el código siguiente:

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,
};
```

En el código anterior se muestra la cláusula `when` de un segmento modificador. Puede usar la cláusula `when` para probar condiciones distintas de la igualdad de una propiedad. Cuando termine, tendrá un método bastante similar al siguiente:

```
vehicle switch
{
    Car { Passengers: 0}        => 2.00m + 0.50m,
    Car { Passengers: 1}        => 2.0m,
    Car { Passengers: 2}        => 2.0m - 0.50m,
    Car c                       => 2.00m - 1.0m,

    Taxi { Fares: 0}  => 3.50m + 1.00m,
    Taxi { Fares: 1 } => 3.50m,
    Taxi { Fares: 2}  => 3.50m - 0.50m,
    Taxi t            => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
    Bus b => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,

    { }     => throw new ArgumentException(message: "Not a known vehicle type", paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};
```

Muchos de estos segmentos modificadores son ejemplos de **patrones recursivos**. Por ejemplo, `Car { Passengers: 1}` muestra un patrón constante dentro de un patrón de propiedad.

Puede usar modificadores anidados para que este código sea menos repetitivo. Tanto `Car` como `Taxi` tienen cuatro segmentos distintos en los ejemplos anteriores. En ambos casos, puede crear un patrón de tipo que se agrega a un patrón de propiedad. Esta técnica se muestra en el código siguiente:

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m + 2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m - 1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle type", paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

En el ejemplo anterior, el uso de una expresión recursiva significa que no repite los segmentos `Car` y `Taxi` que contienen segmentos secundarios que prueban el valor de propiedad. Esta técnica no se usa para los segmentos `Bus` y `DeliveryTruck`, porque esos segmentos prueban intervalos para la propiedad, no valores discretos.

Incorporación de precio en horas punta

Para la característica final, la autoridad encargada de los peajes quiere agregar precios en función de las horas punta. En las horas de mayor afluencia durante mañana y tarde, el valor de los peajes se dobla. Esa regla solo afecta el tráfico en una dirección: hacia la ciudad en la mañana y desde la ciudad en la tarde. En otros momentos durante la jornada laboral, los peajes aumentan en un 50 %. Tarde por la noche y temprano en la mañana, disminuyen en un 25 %. Durante el fin de semana, la tarifa es normal independientemente de la hora.

Para esta característica, usará la coincidencia de patrones, pero la integrará con otras técnicas. Podría crear una expresión de coincidencia de patrones única que consideraría todas las combinaciones de dirección, día de la semana y hora. El resultado sería una expresión complicada. Podría ser difícil de leer y de comprender. Esto implica que es difícil garantizar su exactitud. En su lugar, combine esos método para crear una tupla de valores que describa de manera concisa todos esos estados. Luego, use la coincidencia de patrones para calcular un multiplicador para el peaje. La tupla contiene tres condiciones discretas:

- El día es un día laborable o fin de semana.
- La banda de tiempo cuando se cobra el peaje.
- La dirección si va hacia la ciudad o desde la ciudad.

En la tabla siguiente se muestran las combinaciones de valores de entrada y el multiplicador de precio en horas punta:

DAY	TIEMPO	DIRECCIÓN	RECARGO
Día laborable	hora punta de la mañana	hacia la ciudad	x 2,00
Día laborable	hora punta de la mañana	desde la ciudad	x 1,00
Día laborable	día	hacia la ciudad	x 1,50
Día laborable	día	desde la ciudad	x 1,50
Día laborable	hora punta de la tarde	hacia la ciudad	x 1,00
Día laborable	hora punta de la tarde	desde la ciudad	x 2,00
Día laborable	noche	hacia la ciudad	x 0,75
Día laborable	noche	desde la ciudad	x 0,75
Fin de semana	hora punta de la mañana	hacia la ciudad	x 1,00
Fin de semana	hora punta de la mañana	desde la ciudad	x 1,00
Fin de semana	día	hacia la ciudad	x 1,00
Fin de semana	día	desde la ciudad	x 1,00
Fin de semana	hora punta de la tarde	hacia la ciudad	x 1,00
Fin de semana	hora punta de la tarde	desde la ciudad	x 1,00
Fin de semana	noche	hacia la ciudad	x 1,00
Fin de semana	noche	desde la ciudad	x 1,00

Hay 16 combinaciones distintas de las tres variables. Mediante la combinación de algunas de las condiciones, simplificará la expresión switch final.

El sistema que cobra los peajes usa una estructura DateTime para la hora en que se cobró el peaje. Genere métodos de miembro que creen las variables a partir de la tabla anterior. La función siguiente usa una expresión switch de coincidencia de patrones para expresar si la estructura DateTime representa un día laborable o un fin de semana:

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

Ese método funciona, pero es redundante. Puede simplificarlo tal como se muestra en el código siguiente:

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday   => false,
        _                  => true
    };
```

A continuación, agregue una función similar para categorizar la hora en los bloques:

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll)
{
    int hour = timeOfToll.Hour;
    if (hour < 6)
        return TimeBand.Overnight;
    else if (hour < 10)
        return TimeBand.MorningRush;
    else if (hour < 16)
        return TimeBand.Daytime;
    else if (hour < 20)
        return TimeBand.EveningRush;
    else
        return TimeBand.Overnight;
}
```

El método anterior no usa la coincidencia de patrones. Es más claro usar una cascada familiar de instrucciones `if`. Sí agrega una `enum` privada para convertir cada intervalo de tiempo en un valor discreto.

Después de usar esos métodos, puede usar otra expresión `switch` con el **patrón de tuplas** para calcular el recargo en los precios. Puede crear una expresión `switch` con los 16 segmentos:

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true,  TimeBand.MorningRush, true)  => 2.00m,
        (true,  TimeBand.MorningRush, false) => 1.00m,
        (true,  TimeBand.Daytime,     true)  => 1.50m,
        (true,  TimeBand.Daytime,     false) => 1.50m,
        (true,  TimeBand.EveningRush, true)  => 1.00m,
        (true,  TimeBand.EveningRush, false) => 2.00m,
        (true,  TimeBand.Overnight,   true)  => 0.75m,
        (true,  TimeBand.Overnight,   false) => 0.75m,
        (false, TimeBand.MorningRush, true)  => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime,     true)  => 1.00m,
        (false, TimeBand.Daytime,     false) => 1.00m,
        (false, TimeBand.EveningRush, true)  => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight,   true)  => 1.00m,
        (false, TimeBand.Overnight,   false) => 1.00m,
    };
```

El código anterior funciona, pero se puede simplificar. Las ocho combinaciones para el fin de semana tienen el mismo peaje. Puede reemplazar las ocho por la siguiente línea:

```
(false, _, _) => 1.0m,
```

Tanto el tráfico hacia la ciudad como el tráfico desde la ciudad tienen el mismo multiplicador durante el día y la noche de los fines de semana. Esos cuatro segmentos modificadores se pueden reemplazar por las dos líneas siguientes:

```
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

El código debe ser similar al código siguiente después de esos dos cambios:

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

Por último, puede quitar las dos horas punta en que se paga el precio regular. Cuando quite esos segmentos, puede reemplazar `false` por un descarte (`_`) en el segmento modificador final. Tendrá el siguiente método finalizado:

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (true, TimeBand.Daytime,     _)     => 1.5m,
        (true, TimeBand.MorningRush, true)  => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        (_,    _,                    _)     => 1.0m,
    };
```

En este ejemplo se resalta una de las ventajas de la coincidencia de patrones: las ramas del patrón se evalúan en orden. Si vuelve a ordenarlas para que una rama anterior controle uno de los últimos casos, el compilador genera una advertencia sobre el código inaccesible. Esas reglas de lenguaje facilitan la realización de las simplificaciones anteriores con la confianza de que el código no cambió.

La coincidencia de patrones hace que algunos tipos de código sean más legibles y ofrece una alternativa a las técnicas orientadas a objetos cuando no se puede agregar código a las clases. La nube hace que los datos y la funcionalidad residan por separado. La *forma* de los datos y las *operaciones* que se realizan en ellos no necesariamente se describen en conjunto. En este tutorial, consumió datos existentes de maneras totalmente distintas de su función original. La coincidencia de patrones le ha brindado la capacidad de escribir una funcionalidad que reemplazase a esos tipos, aunque no le permitía extenderlos.

Pasos siguientes

Puede descargar el código finalizado del repositorio GitHub dotnet/samples. Explore los patrones por su cuenta y agregue esta técnica a sus actividades habituales de codificación. Aprender estas técnicas le permite contar con otra forma de enfocar los problemas y crear una funcionalidad nueva.

Aplicación de consola

29/06/2020 • 23 minutes to read • Edit Online

Este tutorial le enseña varias características de .NET Core y el lenguaje C#. Aprenderá lo siguiente:

- Los aspectos básicos de la CLI de .NET Core
- La estructura de una aplicación de consola en C#
- E/S de consola
- Aspectos básicos de las API de E/S de archivo en .NET
- Aspectos básicos de la programación asincrónica basada en tareas en .NET

Creará una aplicación que lea un archivo de texto y refleje el contenido de ese archivo de texto en la consola. El ritmo de la salida a la consola se ajusta para que coincida con la lectura en voz alta. Para aumentar o reducir el ritmo, presione las teclas "<" (menor que) o ">" (mayor que).

Hay muchas características en este tutorial. Vamos a compilarlas una a una.

Requisitos previos

- Configure la máquina para ejecutar .NET Core. Puede encontrar las instrucciones de instalación en la página Descargas de .NET Core. Puede ejecutar esta aplicación en Windows, Linux, macOS o en un contenedor de Docker.
- Instale su editor de código favorito.

Creación de la aplicación

El primer paso es crear una nueva aplicación. Abra un símbolo del sistema y cree un nuevo directorio para la aplicación. Conviértalo en el directorio actual. Escriba el comando `dotnet new console` en el símbolo del sistema. Esta acción crea los archivos de inicio para una aplicación básica "Hola mundo".

Antes de comenzar a realizar modificaciones, vamos a recorrer los pasos para ejecutar la aplicación Hola mundo sencilla. Después de crear la aplicación, escriba `dotnet restore` en el símbolo del sistema. Este comando ejecuta el proceso de restauración de paquetes de NuGet. NuGet es un administrador de paquetes .NET. Este comando permite descargar cualquiera de las dependencias que faltan para el proyecto. Como se trata de un nuevo proyecto, ninguna de las dependencias está en su lugar, así que con la primera ejecución se descargará .NET Core Framework. Después de este paso inicial, solo deberá ejecutar `dotnet restore` al agregar nuevos paquetes dependientes, o actualizar las versiones de cualquiera de sus dependencias.

No es necesario ejecutar `dotnet restore` porque lo ejecutan implícitamente todos los comandos que necesitan que se produzca una restauración, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` y `dotnet pack`. Para deshabilitar la restauración implícita, use la opción `--no-restore`.

El comando `dotnet restore` sigue siendo válido en algunos escenarios donde tiene sentido realizar una restauración explícita, como las compilaciones de integración continua en Azure DevOps Services o en los sistemas de compilación que necesitan controlar explícitamente cuándo se produce la restauración.

Para obtener información sobre cómo administrar fuentes de NuGet, vea la documentación de `dotnet restore`.

Después de restaurar los paquetes, ejecutará `dotnet build`. Esta acción ejecuta el motor de compilación y crea el ejecutable de aplicación. Por último, ejecute `dotnet run` para ejecutar la aplicación.

El código de la aplicación sencilla Hola a todos está todo en Program.cs. Abra ese archivo con el editor de texto de su elección. Nos disponemos a realizar nuestros primeros cambios. En la parte superior del archivo, verá una instrucción using:

```
using System;
```

Esta instrucción indica al compilador que cualquier tipo del espacio de nombres `System` está en el ámbito. Al igual que otros lenguajes orientados a objetos que pueda haber usado, C# utiliza espacios de nombres para organizar los tipos. Este programa Hola mundo no es diferente. Puede ver que el programa se incluye en el espacio de nombres y el nombre se basa en el del directorio actual. Para este tutorial, vamos a cambiar el nombre del espacio de nombres por `TeleprompterConsole`:

```
namespace TeleprompterConsole
```

Lectura y reflejo del archivo

La primera característica que se va a agregar es la capacidad de leer un archivo de texto y visualizar todo ese texto en la consola. En primer lugar, vamos a agregar un archivo de texto. Copie el archivo sampleQuotes.txt del repositorio de GitHub de este ejemplo en su directorio del proyecto. Servirá como script para la aplicación. Si quiere obtener información sobre cómo descargar la aplicación de ejemplo de este tema, vea las instrucciones del tema Ejemplos y tutoriales.

Luego, agregue el siguiente método a la clase `Program` (justo debajo del método `Main`):

```
static IEnumerable<string> ReadFrom(string file)
{
    string line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

Este método usa tipos de dos nuevos espacios de nombres. Para compilar esto, deberá agregar las dos líneas siguientes a la parte superior del archivo:

```
using System.Collections.Generic;
using System.IO;
```

La interfaz IEnumerable<T> se define en el espacio de nombres System.Collections.Generic. La clase File se define en el espacio de nombres System.IO.

Este método es un tipo especial de método de C# llamado *método de iterador*. Los métodos de enumerador devuelven secuencias que se evalúan de forma diferida. Eso significa que cada elemento de la secuencia se genera como lo solicita el código que consume la secuencia. Los métodos de enumerador son métodos que contienen una o varias instrucciones `yield return`. El objeto que devuelve el método `ReadFrom` contiene el código para generar cada elemento en la secuencia. En este ejemplo, que implica la lectura de la siguiente línea de texto del archivo de origen y la devolución de esa cadena, cada vez que el código de llamada solicita el siguiente elemento de la secuencia, el código lee la siguiente línea de texto del archivo y la devuelve. Cuando el archivo se ha leído completamente, la secuencia indica que no hay más elementos.

Hay otros dos elementos de la sintaxis de C# con los que podría no estar familiarizado. La instrucción `using` de este método administra la limpieza de recursos. La variable que se inicializa en la instrucción `using` (`reader`, en este ejemplo) debe implementar la interfaz IDisposable. Esa interfaz define un único método, `Dispose`, que se debe llamar cuando sea necesario liberar el recurso. El compilador genera esa llamada cuando la ejecución llega a la llave de cierre de la instrucción `using`. El código generado por el compilador garantiza que el recurso se libera incluso si se produce una excepción desde el código en el bloqueo definido mediante la instrucción using.

La variable `reader` se define mediante la palabra clave `var`. `var` define una *variable local con tipo implícito*. Esto significa que el tipo de la variable viene determinado por el tipo en tiempo de compilación del objeto asignado a la variable. Aquí, ese es el valor devuelto por el método OpenText(String), que es un objeto StreamReader.

Ahora, vamos a rellenar el código para leer el archivo en el método `Main`:

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Ejecute el programa (mediante `dotnet run`) y podrá ver cada línea impresa en la consola.

Adición de retrasos y formato de salida

Lo que tiene se va a mostrar lejos, demasiado rápido para leerlo en voz alta. Ahora debe agregar los retrasos en la salida. Cuando empiece, estará creando parte del código principal que permite el procesamiento asincrónico. Sin embargo, a estos primeros pasos le seguirán algunos antipatrones. Los antipatrones se señalan en los comentarios cuando se agrega el código y el código se actualizará en los pasos posteriores.

Hay dos pasos hasta esta sección. Primero, actualizará el método Iterator para devolver palabras sueltas en lugar de líneas enteras. Para ello, son necesarias estas modificaciones. Reemplace la instrucción `yield return line;` por el código siguiente:

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

A continuación, debe modificar el modo en que se consumen las líneas del archivo y agregar un retraso después de escribir cada palabra. Reemplace la instrucción `Console.WriteLine(line)` del método `Main` por el bloqueo siguiente:

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

La clase Task está en el espacio de nombres System.Threading.Tasks, así que debe agregar esa instrucción `using` en la parte superior del archivo:

```
using System.Threading.Tasks;
```

Ejecute el ejemplo y compruebe la salida. Ahora, se imprime cada palabra suelta, seguido de un retraso de 200 ms. Sin embargo, la salida mostrada indica algunos problemas porque el archivo de texto de origen tiene varias líneas con más de 80 caracteres sin un salto de línea. Este texto puede ser difícil de leer al desplazarse por él. Esto es fácil de corregir. Simplemente realizará el seguimiento de la longitud de cada línea y generará una nueva línea cada vez que la longitud de la línea alcance un determinado umbral. Declare una variable local después de la declaración de `words` en el método `ReadFrom` que contiene la longitud de línea:

```
var lineLength = 0;
```

A continuación, agregue el código siguiente después de la instrucción `yield return word + " ";` (antes de la llave de cierre):

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

Ejecute el ejemplo y podrá leer en alto a su ritmo preconfigurado.

Tareas asincrónicas

En este paso final, agregará el código para escribir la salida de manera asincrónica en una tarea, mientras se ejecuta también otra tarea para leer la entrada del usuario si quiere aumentar o reducir la velocidad de la pantalla de texto, o detendrá la presentación del texto por completo. Incluye unos cuantos pasos y, al final, tendrá todas las actualizaciones que necesita. El primer paso es crear un método de devolución Task asincrónico que represente el código que ha creado hasta el momento para leer y visualizar el archivo.

Agregue este método a su clase `Program` (se toma del cuerpo del método `Main`):

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

Advertirá dos cambios. Primero, en el cuerpo del método, en lugar de llamar a Wait() para esperar a que finalice una tarea de manera sincrónica, esta versión usa la palabra clave `await`. Para ello, debe agregar el modificador `async` a la signatura del método. Este método devuelve un objeto `Task`. Observe que no hay ninguna instrucción Return que devuelva un objeto `Task`. En su lugar, ese objeto `Task` se crea mediante el código que genera el compilador cuando usa el operador `await`. Puede imaginar que este método devuelve cuando alcanza un valor de `await`. El valor devuelto de `Task` indica que el trabajo no ha finalizado. El método se reanuda cuando se completa la tarea en espera. Cuando se ha ejecutado hasta su finalización, el valor de `Task` devuelto indica que se ha completado. El código de llamada puede supervisar ese valor de `Task` devuelto para determinar cuándo se ha

completado.

Puede llamar a este nuevo método en su método `Main`:

```
ShowTeleprompter().Wait();
```

Aquí, en `Main`, el código espera de manera sincrónica. Siempre que sea posible, debe usar el operador `await` en lugar de esperar sincrónicamente. Sin embargo, en el método `Main` de una aplicación de consola, no puede usar el operador `await`. Si así fuera, la aplicación se cerraría antes de que todas las tareas se hubieran completado.

> **NOTE**
> Si usa C# 7.1 o una versión posterior, puede crear aplicaciones de consola con el método `async` `Main`.

A continuación, debe escribir el segundo método asincrónico para leer de la consola y controlar las teclas "<" (menor que), ">" (mayor que), "X" o "x". Este es el método que agrega para esa tarea:

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

Se crea una expresión lambda que representa un delegado de Action que lee una clave de la consola y modifica una variable local que representa el retraso cuando el usuario presiona las teclas "<" (menor que) o ">" (mayor que). El método de delegado finaliza cuando el usuario presiona las teclas "X" o "x", que permiten al usuario detener la presentación del texto en cualquier momento. Este método usa ReadKey() para bloquear y esperar a que el usuario presione una tecla.

Para finalizar esta característica, debe crear un nuevo método de devolución `async Task` que inicie estas dos tareas (`GetInput` y `ShowTeleprompter`) y también administre los datos compartidos entre ellas.

Es hora de crear una clase que controle los datos compartidos entre estas dos tareas. Esta clase contiene dos propiedades públicas: el retraso y una marca `Done` para indicar que el archivo se ha leído completamente:

```
namespace TeleprompterConsole
{
    internal class TelePrompterConfig
    {
        public int DelayInMilliseconds { get; private set; } = 200;

        public void UpdateDelay(int increment) // negative to speed up
        {
            var newDelay = Min(DelayInMilliseconds + increment, 1000);
            newDelay = Max(newDelay, 20);
            DelayInMilliseconds = newDelay;
        }

        public bool Done { get; private set; }

        public void SetDone()
        {
            Done = true;
        }
    }
}
```

Coloque esa clase en un archivo nuevo e inclúyala en el espacio de nombres `TeleprompterConsole`, como se ha mostrado anteriormente. También deberá agregar una instrucción `using static` para que pueda hacer referencia a los métodos `Min` y `Max` sin la clase incluida o los nombres de espacio de nombres. Una instrucción `using static` importa los métodos de una clase, mientras que las instrucciones `using` usadas hasta el momento han importado todas las clases de un espacio de nombres.

```
using static System.Math;
```

A continuación, debe actualizar los métodos `ShowTeleprompter` y `GetInput` para usar el nuevo objeto `config`. Escriba un método final `async` de devolución de `Task` para iniciar ambas tareas y salir cuando la primera tarea finalice:

```
private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

El método nuevo aquí es la llamada a WhenAny(Task[]). Dicha llamada crea un valor de `Task` que finaliza en cuanto alguna de las tareas de su lista de argumentos se completa.

A continuación, debe actualizar los métodos `ShowTeleprompter` y `GetInput` para usar el objeto `config` para el retraso:

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

Esta nueva versión de `ShowTeleprompter` llama a un nuevo método de la clase `TelepromptерConfig`. Ahora, debe actualizar `Main` para llamar a `RunTeleprompter` en lugar de a `ShowTeleprompter`:

```
RunTeleprompter().Wait();
```

Conclusión

En este tutorial se han mostrado varias características en torno al lenguaje C# y las bibliotecas .NET Core, relacionadas con el trabajo en aplicaciones de consola. Puede partir de este conocimiento para explorar más sobre el lenguaje y las clases aquí presentadas. Ha visto los conceptos básicos de E/S de archivo y consola, el uso con bloqueo y sin bloqueo de la programación asincrónica basada en tareas, un paseo por el lenguaje C# y cómo se organizan los programas en C#. También ha conocido la interfaz de la línea de comandos y la CLI de .NET Core.

Para obtener más información sobre la E/S de archivo, consulte el tema E/S de archivos y secuencias. Para obtener más información sobre el modelo de programación asincrónica que se ha usado en este tutorial, vea los temas Programación asincrónica basada en tareas y Programación asincrónica.

Cliente REST

29/06/2020 • 22 minutes to read • Edit Online

Este tutorial le enseña varias características de .NET Core y el lenguaje C#. Aprenderá lo siguiente:

- Los aspectos básicos de la CLI de .NET Core.
- Información general de las características del lenguaje C#.
- Administración de dependencias con NuGet
- Comunicaciones HTTP
- Procesamiento de información de JSON
- Administración de la configuración con atributos

Creará una aplicación que emite solicitudes HTTP a un servicio REST en GitHub. Leerá información en formato JSON y convertirá ese paquete JSON en objetos de C#. Por último, verá cómo trabajar con objetos de C#.

Este tutorial incluye muchas características. Vamos a compilarlas una a una.

Si prefiere seguir el ejemplo final de este tema, puede descargarlo. Para obtener instrucciones de descarga, vea Ejemplos y tutoriales.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core. Puede encontrar las instrucciones de instalación en la página Descargas de .NET Core. Puede ejecutar esta aplicación en Windows, Linux, Mac OS o en un contenedor de Docker. Deberá instalar su editor de código favorito. En las siguientes descripciones se usa Visual Studio Code, que es un editor multiplataforma de código abierto. Sin embargo, puede usar las herramientas que le resulten más cómodas.

Crear la aplicación

El primer paso es crear una nueva aplicación. Abra un símbolo del sistema y cree un nuevo directorio para la aplicación. Conviértalo en el directorio actual. Escriba el siguiente comando en la ventana de consola:

```
dotnet new console --name WebAPIClient
```

Esta acción crea los archivos de inicio para una aplicación básica "Hola mundo". El nombre del proyecto es "WebAPIClient". Como se trata de un proyecto nuevo, ninguna de las dependencias está en su lugar, así que la primera ejecución descargará .NET Core Framework, instalará un certificado de desarrollo y ejecutará el gestor de paquetes NuGet para restaurar las dependencias que faltan.

Antes de empezar a hacer modificaciones, escriba `dotnet run` (vea la nota) en el símbolo del sistema para ejecutar la aplicación. `dotnet run` ejecuta automáticamente `dotnet restore` si el entorno no tiene dependencias. También ejecuta `dotnet build` si hay que volver a compilar la aplicación. Después de la instalación inicial, solo tendrá que ejecutar `dotnet restore` o `dotnet build` cuando tenga sentido para su proyecto.

Adición de nuevas dependencias

Uno de los principales objetivos de diseño de .NET Core es minimizar el tamaño de la instalación de .NET. Si una aplicación necesita bibliotecas adicionales para algunas de sus características, agregará esas dependencias al archivo de proyecto (*.csproj) de C#. En nuestro ejemplo, deberá agregar el paquete

`System.Runtime.Serialization.Json` para que su aplicación pueda procesar respuestas JSON.

Necesitará el paquete `System.Runtime.Serialization.Json` para esta aplicación. Agréguelo al proyecto ejecutando el siguiente comando de la CLI de .NET:

```
dotnet add package System.Text.Json
```

Realización de las solicitudes web

Ahora está listo para comenzar a recuperar datos de la Web. En esta aplicación, leerá información de la API de GitHub. Vamos a leer información sobre los proyectos que aparecen en el paraguas .NET Foundation. Para comenzar, realizará la solicitud a la API de GitHub para recuperar información sobre los proyectos. El punto de conexión que usará es: https://api.github.com/orgs/dotnet/repos. Quiere recuperar toda la información sobre estos proyectos, así que usará una solicitud HTTP GET. El explorador también usa solicitudes HTTP GET, por lo que puede pegar esa dirección URL en él para ver la información que recibirá y procesará.

Usará la clase HttpClient para realizar solicitudes web. Al igual que las API de .NET modernas, HttpClient admite solo métodos asincrónicos para sus API de ejecución prolongada. Para empezar, cree un método asincrónico. Rellenará la implementación mientras compila la funcionalidad de la aplicación. Para comenzar, abra el archivo `program.cs` en el directorio del proyecto y agregue el siguiente método a la clase `Program`:

```
private static async Task ProcessRepositories()
{
}
```

Tiene que agregar una directiva `using` en la parte superior del método `Main` para que el compilador de C# reconozca el tipo Task:

```
using System.Threading.Tasks;
```

Si compila el proyecto en este momento, obtendrá una advertencia generada para este método, porque no contiene ningún operador `await` y se ejecutará de forma sincrónica. Por ahora omítala; agregará los operadores `await` a medida que rellena el método.

Luego, actualice el método `Main` para llamar al método `ProcessRepositories`. El método `ProcessRepositories` devuelve una tarea. No debe salir del programa antes de que esta finalice. Por lo tanto, debe cambiar la firma de `Main`. Agregue el modificador `async` y cambie el tipo de valor devuelto a `Task`. A continuación, en el cuerpo del método, agregue una llamada a `ProcessRepositories`. Agregue la palabra clave `await` a esa llamada al método:

```
static async Task Main(string[] args)
{
    await ProcessRepositories();
}
```

Ahora, tiene un programa que no hace nada, pero lo hace de forma asincrónica. Vamos a mejorarlo.

Primero necesita un objeto capaz de recuperar datos de la Web; para ello, puede usar HttpClient. Este objeto administra la solicitud y las respuestas. Cree una sola instancia de ese tipo en la clase `Program`, dentro del archivo *Program.cs*.

```
namespace WebAPIClient
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static async Task Main(string[] args)
        {
            //...
        }
    }
}
```

Volvamos al método `ProcessRepositories` y rellene una primera versión de él:

```
private static async Task ProcessRepositories()
{
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
    client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository Reporter");

    var stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.Write(msg);
}
```

También tiene que agregar dos nuevas directivas `using` en la parte superior del archivo para que este se compile:

```
using System.Net.Http;
using System.Net.Http.Headers;
```

Esta primera versión realiza una solicitud web para leer la lista de todos los repositorios en la organización dotnet foundation. (El id. de gitHub para .NET Foundation es "dotnet"). Las primeras líneas configuran el objeto HttpClient para esta solicitud. En primer lugar, se configura para aceptar las respuestas JSON de GitHub. Este formato es simplemente JSON. La siguiente línea agrega un encabezado de agente de usuario a todas las solicitudes desde este objeto. Estos dos encabezados se comprueban mediante el código de servidor de GitHub y son necesarios para recuperar información de GitHub.

Después de haber configurado el objeto HttpClient, realizará una solicitud web y recuperará la respuesta. En esta primera versión, usa el método de conveniencia HttpClient.GetStringAsync(String). Este método de conveniencia inicia una tarea que realiza la solicitud web y, a continuación, cuando la solicitud se devuelve, lee la secuencia de respuesta y extrae el contenido de la secuencia. El cuerpo de la respuesta se devuelve como String. La cadena está disponible cuando finaliza la tarea.

Las dos últimas líneas de este método esperan a la tarea y, luego, imprimen la respuesta en la consola. Compile la aplicación y ejecútela. La advertencia de compilación desaparece ahora, porque `ProcessRepositories` contiene ahora un operador `await`. Verá una pantalla larga de texto con formato de JSON.

Procesamiento del resultado JSON

Llegados a este punto, ha escrito el código para recuperar una respuesta de un servidor web y mostrar el texto contiene esa respuesta. A continuación, vamos a convertir esa respuesta JSON en objetos de C#.

La clase System.Text.Json.JsonSerializer serializa objetos a JSON y deserializa JSON en objetos. Empiece por definir una clase para representar el objeto JSON `repo` devuelto desde la API de GitHub:

```
using System;

namespace WebAPIClient
{
    public class Repository
    {
        public string name { get; set; }
    }
}
```

Coloque el código anterior en un archivo nuevo llamado "repo.cs". Esta versión de la clase representa la ruta de acceso más sencilla de procesar datos JSON. El nombre de clase y el nombre de miembro coinciden con los nombres usados en el paquete JSON, en lugar de las siguientes convenciones de C#. Más adelante proporcionará algunos atributos de configuración para corregir esto. Esta clase muestra otra característica importante de la serialización y deserialización de JSON: No todos los campos del paquete JSON forman parte de esta clase. El serializador JSON pasará por alto la información que no esté incluida en el tipo de clase que se va a usar. Esta característica facilita la creación de tipos que funcionan con solo un subconjunto de los campos del paquete JSON.

Ahora que ha creado el tipo, vamos a deserializarlo.

A continuación, usará el serializador para convertir JSON en objetos de C#. Reemplace la llamada a GetStringAsync(String) en su método `ProcessRepositories` por las líneas siguientes:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
```

Está usando un nuevo espacio de nombres, por lo que deberá agregarlo también en la parte superior del archivo:

```
using System.Collections.Generic;
using System.Text.Json;
```

Tenga en cuenta que ahora usa GetStreamAsync(String) en lugar de GetStringAsync(String). El serializador usa una secuencia en lugar de una cadena como su origen. Vamos a explicar algunas características del lenguaje C# que se usan en la segunda línea del fragmento de código anterior. El primer argumento para JsonSerializer.DeserializeAsync<TValue>(Stream, JsonSerializerOptions, CancellationToken) es una expresión `await`. Los otros dos parámetros son opcionales y se omiten en el fragmento de código. Las expresiones Await pueden aparecer prácticamente en cualquier parte del código, aunque hasta ahora, únicamente las ha visto como parte de una instrucción de asignación. El método `Deserialize` es *genérico*, lo que significa que debe proporcionar argumentos de tipo para el tipo de objetos que se debe crear a partir del texto JSON. En este ejemplo, se va a deserializar a un `List<Repository>`, que es otro objeto genérico, System.Collections.Generic.List<T>. La clase `List<>` administra una colección de objetos. El argumento de tipo declara el tipo de objetos almacenados en `List<>`. El texto JSON representa una colección de objetos de repositorio, por lo que el argumento de tipo es `Repository`.

Casi ha terminado con esta sección. Ahora que ha convertido el código JSON a objetos de C#, vamos a mostrar el nombre de cada repositorio. Reemplace las líneas donde pone:

```
var msg = await stringTask;   //**Deleted this
Console.Write(msg);
```

por lo siguiente:

```
foreach (var repo in repositories)
    Console.WriteLine(repo.name);
```

Compile y ejecute la aplicación. Se imprimirán los nombres de los repositorios que forman parte de .NET Foundation.

Control de la serialización

Antes de agregar más características, vamos a abordar la propiedad `name` mediante el atributo `[JsonPropertyName]`. Realice los siguientes cambios en la declaración del campo `name` en repo.cs:

```
[JsonPropertyName("name")]
public string Name { get; set; }
```

Para usar el atributo `[JsonPropertyName]`, tiene que agregar el espacio de nombres System.Text.Json.Serialization a las directivas `using`:

```
using System.Text.Json.Serialization;
```

Este cambio significa que debe cambiar el código que escribe el nombre de cada repositorio en program.cs:

```
Console.WriteLine(repo.Name);
```

Ejecute `dotnet run` para asegurarse de que tiene las asignaciones correctas. Debería ver la misma salida que antes.

Vamos a hacer un cambio más antes de agregar nuevas características. El método `ProcessRepositories` puede realizar el trabajo asincrónico y devolver una colección de los repositorios. Vamos a volver a `List<Repository>` desde ese método y a mover el código que escribe la información en el método `Main`.

Cambie la signatura de `ProcessRepositories` para devolver una tarea cuyo resultado sea una lista de objetos `Repository`:

```
private static async Task<List<Repository>> ProcessRepositories()
```

A continuación, devuelva solo los repositorios después de procesar la respuesta JSON:

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
return repositories;
```

El compilador genera el objeto `Task<T>` para la devolución porque ha marcado este método como `async`. A continuación, vamos a modificar el método `Main` para que capture esos resultados y escriba cada nombre de repositorio en la consola. Su método `Main` tiene ahora este aspecto:

```
public static async Task Main(string[] args)
{
    var repositories = await ProcessRepositories();

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

Leer más información

Para terminar esto, vamos a procesar unas cuantas propiedades más del paquete JSON que se envía desde la API de GitHub. No quiere captar toso, sino que con solo agregar unas cuantas propiedades se demostrarán más características del lenguaje C#.

Comencemos por agregar algunos tipos simples más a la definición de clase `Repository`. Agregue estas propiedades a esa clase:

```
[JsonPropertyName("description")]
public string Description { get; set; }

[JsonPropertyName("html_url")]
public Uri GitHubHomeUrl { get; set; }

[JsonPropertyName("homepage")]
public Uri Homepage { get; set; }

[JsonPropertyName("watchers")]
public int Watchers { get; set; }
```

Estas propiedades tienen integradas conversiones de tipo cadena (que es lo que contienen los paquetes JSON) para el tipo de destino. Puede que no esté familiarizado con el tipo Uri. Representa un identificador URI, o en este caso, una dirección URL. En el caso de los tipos `Uri` y `int`, si el paquete JSON contiene datos que no se convierten al tipo de destino, la acción de serialización iniciará una excepción.

Una vez agregados estos, actualice el método `Main` para mostrar estos elementos:

```
foreach (var repo in repositories)
{
    Console.WriteLine(repo.Name);
    Console.WriteLine(repo.Description);
    Console.WriteLine(repo.GitHubHomeUrl);
    Console.WriteLine(repo.Homepage);
    Console.WriteLine(repo.Watchers);
    Console.WriteLine();
}
```

Como paso final, vamos a agregar la información de la última operación de inserción. El formato de esta información es este en la respuesta JSON:

```
2016-02-08T21:27:00Z
```

Ese formato está en hora universal coordinada (UTC), por lo que obtendrá un valor DateTime cuya propiedad Kind es Utc. Si prefiere una fecha representada en su zona horaria, deberá escribir un método de conversión personalizado. En primer lugar, defina una propiedad `public` que contendrá la representación UTC de la fecha y hora en la clase `Repository` y una propiedad `LastPush` `readonly` que devuelve la fecha convertida en la hora local:

```
[JsonPropertyName("pushed_at")]
public DateTime LastPushUtc { get; set; }

public DateTime LastPush => LastPushUtc.ToLocalTime();
```

Vamos a repasar las nuevas construcciones que acabamos de definir. La propiedad `LastPush` se define utilizando un *miembro con forma de expresión* para el descriptor de acceso `get`. No hay descriptor de acceso `set`. Omitiendo el descriptor de acceso `set` es la forma en que se define una propiedad *de solo lectura* en C#. (Sí, puede crear propiedades de *solo escritura* en C#, pero su valor es limitado).

Finalmente, agregue una instrucción más de salida en la consola y ya estará listo para compilar y ejecutar de nuevo esta aplicación:

```
Console.WriteLine(repo.LastPush);
```

La versión debe coincidir ahora con el ejemplo terminado.

Conclusión

En este tutorial se ha mostrado cómo realizar solicitudes web, analizar el resultado y visualizar las propiedades de los resultados. También ha agregado nuevos paquetes como dependencias en el proyecto. Ha visto algunas de las características del lenguaje C# compatibles con técnicas orientadas a objetos.

No es necesario ejecutar `dotnet restore` porque lo ejecutan implícitamente todos los comandos que necesitan que se produzca una restauración, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` y `dotnet pack`. Para deshabilitar la restauración implícita, use la opción `--no-restore`.

El comando `dotnet restore` sigue siendo válido en algunos escenarios donde tiene sentido realizar una restauración explícita, como las compilaciones de integración continua en Azure DevOps Services o en los sistemas de compilación que necesitan controlar explícitamente cuándo se produce la restauración.

Para obtener información sobre cómo administrar fuentes de NuGet, vea la documentación de `dotnet restore`.

Herencia en C# y .NET

29/06/2020 • 46 minutes to read • Edit Online

Este tutorial es una introducción a la herencia en C#. La herencia es una característica de los lenguajes de programación orientados a objetos que permite definir una clase base, que proporciona funcionalidad específica (datos y comportamiento), así como clases derivadas, que heredan o invalidan esa funcionalidad.

Requisitos previos

En este tutorial se supone que ha instalado el SDK de .NET Core. Visite la página de descargas de .NET Core para descargarlo. También necesitará un editor de código. En este tutorial se usa Visual Studio Code, aunque puede elegir el que prefiera.

Ejecución de los ejemplos

Para crear y ejecutar los ejemplos de este tutorial, use la utilidad dotnet desde la línea de comandos. Siga estos pasos para cada ejemplo:

1. Cree un directorio para almacenar el ejemplo.
2. Escriba el comando dotnet new console en el símbolo del sistema para crear un nuevo proyecto de .NET Core.
3. Copie y pegue el código del ejemplo en el editor de código.
4. Escriba el comando dotnet restore desde la línea de comandos para cargar o restaurar las dependencias del proyecto.

   No es necesario ejecutar `dotnet restore` porque lo ejecutan implícitamente todos los comandos que necesitan que se produzca una restauración, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` y `dotnet pack`. Para deshabilitar la restauración implícita, use la opción `--no-restore`.

   El comando `dotnet restore` sigue siendo válido en algunos escenarios donde tiene sentido realizar una restauración explícita, como las compilaciones de integración continua en Azure DevOps Services o en los sistemas de compilación que necesitan controlar explícitamente cuándo se produce la restauración.

   Para obtener información sobre cómo administrar fuentes de NuGet, vea la documentación de `dotnet restore`.
5. Escriba el comando dotnet run para compilar y ejecutar el ejemplo.

Información previa: ¿Qué es la herencia?

La *herencia* es uno de los atributos fundamentales de la programación orientada a objetos. Permite definir una clase secundaria que reutiliza (hereda), amplía o modifica el comportamiento de una clase primaria. La clase cuyos miembros son heredados se conoce como *clase base*. La clase que hereda los miembros de la clase base se conoce como *clase derivada*.

C# y .NET solo admiten *herencia única*. Es decir, una clase solo puede heredar de una clase única. Sin embargo, la herencia es transitiva, lo que le permite definir una jerarquía de herencia para un conjunto de tipos. En otras palabras, el tipo `D` puede heredar del tipo `C`, que hereda del tipo `B`, que hereda del tipo de clase base `A`. Dado que la herencia es transitiva, los miembros de tipo `A` están disponibles para el tipo `D`.

No todos los miembros de una clase base los heredan las clases derivadas. Los siguientes miembros no se heredan:

- Constructores estáticos, que inicializan los datos estáticos de una clase.
- Constructores de instancias, a los que se llama para crear una nueva instancia de la clase. Cada clase debe definir sus propios constructores.
- Finalizadores, llamados por el recolector de elementos no utilizados en tiempo de ejecución para destruir instancias de una clase.

Si bien las clases derivadas heredan todos los demás miembros de una clase base, que dichos miembros estén o no visibles depende de su accesibilidad. La accesibilidad del miembro afecta a su visibilidad en las clases derivadas del modo siguiente:

- Los miembros privados solo son visible en las clases derivadas que están anidadas en su clase base. De lo contrario, no son visibles en las clases derivadas. En el ejemplo siguiente, `A.B` es una clase anidada que se deriva de `A`, y `C` se deriva de `A`. El campo `A.value` privado es visible en A.B. Sin embargo, si quita los comentarios del método `C.GetValue` e intenta compilar el ejemplo, se produce el error del compilador CS0122: ""A.value" no es accesible debido a su nivel de protección".

```
using System;

public class A
{
   private int value = 10;

   public class B : A
   {
       public int GetValue()
       {
           return this.value;
       }
   }
}

public class C : A
{
//    public int GetValue()
//    {
//        return this.value;
//    }
}

public class Example
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//       10
```

- Los miembros protegidos solo son visibles en las clases derivadas.
- Los miembros internos solo son visibles en las clases derivadas que se encuentran en el mismo ensamblado que la clase base. No son visibles en las clases derivadas ubicadas en un ensamblado diferente al de la clase base.
- Los miembros públicos son visibles en las clases derivadas y forman parte de la interfaz pública de dichas

clases. Los miembros públicos heredados se pueden llamar como si se definieran en la clase derivada. En el ejemplo siguiente, la clase `A` define un método denominado `Method1` y la clase `B` hereda de la clase `A`. El ejemplo llama a `Method1` como si fuera un método de instancia en `B`.

```
public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new B();
        b.Method1();
    }
}
```

Las clases derivadas pueden también *invalidar* los miembros heredados al proporcionar una implementación alternativa. Para poder invalidar un miembro, el miembro de la clase base debe marcarse con la palabra clave virtual. De forma predeterminada, los miembros de clase base no están marcados con `virtual` y no se pueden invalidar. Al intentar invalidar un miembro no virtual, como en el ejemplo siguiente, se genera el error de compilador CS0506: "<member> no se puede invalidar el miembro heredado <member> porque no está marcado como virtual, abstract ni override.

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

En algunos casos, una clase derivada *debe* invalidar la implementación de la clase base. Los miembros de clase base marcados con la palabra clave abstract requieren que las clases derivadas los invaliden. Al intentar compilar el ejemplo siguiente, se genera el error de compilador CS0534, "<class> no implementa el miembro abstracto heredado <member>", porque la clase `B` no proporciona ninguna implementación para `A.Method1`.

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

La herencia solo se aplica a clases e interfaces. Other type categories (structs, delegates, and enums) do not support inheritance. Debido a estas reglas, al intentar compilar código como el siguiente se genera el error de compilador CS0527: "El tipo 'ValueType' de la lista de interfaces no es una interfaz". El mensaje de error indica que, aunque se pueden definir las interfaces que implementa un struct, no se admite la herencia.

```
using System;

public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

Herencia implícita

Aparte de los tipos de los que puedan heredar mediante herencia única, todos los tipos del sistema de tipos .NET heredan implícitamente de Object o de un tipo derivado de este. La funcionalidad común de Object está disponible para cualquier tipo.

Para ver lo que significa la herencia implícita, vamos a definir una nueva clase, `SimpleClass`, que es simplemente una definición de clase vacía:

```
public class SimpleClass
{ }
```

Después, se puede usar la reflexión (que permite inspeccionar los metadatos de un tipo para obtener información sobre ese tipo) con el fin de obtener una lista de los miembros que pertenecen al tipo `SimpleClass`. Aunque no se ha definido ningún miembro en la clase `SimpleClass`, la salida del ejemplo indica que en realidad tiene nueve miembros. Uno de ellos es un constructor sin parámetros (o predeterminado) que el compilador de C# proporciona de manera automática para el tipo `SimpleClass`. Los ocho restantes son miembros de Object, el tipo del que heredan implícitamente a la larga todas las clases e interfaces del sistema de tipo .NET.

```
using System;
using System.Reflection;

public class Example
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
                             BindingFlags.NonPublic | BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (var member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            var output = $"{member.Name} ({member.MemberType}): {access}{stat}, Declared by
{member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}
// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method):  Public, Declared by System.Object
// Equals (Method):  Public, Declared by System.Object
// Equals (Method):  Public Static, Declared by System.Object
// ReferenceEquals (Method):  Public Static, Declared by System.Object
// GetHashCode (Method):  Public, Declared by System.Object
// GetType (Method):  Public, Declared by System.Object
// Finalize (Method):  Internal, Declared by System.Object
// MemberwiseClone (Method):  Internal, Declared by System.Object
// .ctor (Constructor):  Public, Declared by SimpleClass
```

La herencia implícita desde la clase Object permite que estos métodos estén disponibles para la clase `SimpleClass`:

- El método público `ToString`, que convierte un objeto `SimpleClass` en su representación de cadena, devuelve el nombre de tipo completo. En este caso, el método `ToString` devuelve la cadena "SimpleClass".
- Tres métodos de prueba de igualdad de dos objetos: el método de instancia pública `Equals(Object)`, el método público estático `Equals(Object, Object)` y el método público estático `ReferenceEquals(Object, Object)`. De forma predeterminada, estos métodos prueban la igualdad de referencia; es decir, para que sean iguales, dos variables de objeto deben hacer referencia al mismo objeto.
- El método público `GetHashCode`, que calcula un valor que permite que una instancia del tipo se use en colecciones con hash.
- El método público `GetType`, que devuelve un objeto Type que representa el tipo `SimpleClass`.

- El método protegido Finalize, que está diseñado para liberar recursos no administrados antes de que el recolector de elementos no utilizados reclame la memoria de un objeto.
- El método protegido MemberwiseClone, que crea un clon superficial del objeto actual.

Debido a la herencia implícita, se puede llamar a cualquier miembro heredado de un objeto `SimpleClass` como si realmente fuera un miembro definido en la clase `SimpleClass`. Así, en el ejemplo siguiente se llama al método `SimpleClass.ToString`, que `SimpleClass` hereda de Object.

```
using System;

public class SimpleClass
{}

public class Example
{
    public static void Main()
    {
        SimpleClass sc = new SimpleClass();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//        SimpleClass
```

En la tabla siguiente se enumeran las categorías de tipos que se pueden crear en C# y los tipos de los que heredan implícitamente. Cada tipo base constituye un conjunto diferente de miembros disponible mediante herencia para los tipos derivados de forma implícita.

CATEGORÍA DE TIPO	HEREDA IMPLÍCITAMENTE DE
clase	Object
struct	ValueType, Object
enum	Enum, ValueType, Object
delegado	MulticastDelegate, Delegate, Object

Herencia y una relación "is a"

Normalmente, la herencia se usa para expresar una relación "is a" entre una clase base y una o varias clases derivadas, donde las clases derivadas son versiones especializadas de la clase base; la clase derivada es un tipo de la clase base. Por ejemplo, la clase `Publication` representa una publicación de cualquier tipo y las clases `Book` y `Magazine` representan tipos específicos de publicaciones.

> **NOTE**
> Una clase o struct puede implementar una o varias interfaces. Aunque a menudo la implementación se presenta como una solución alternativa para la herencia única o como una forma de usar la herencia con structs, su finalidad es expresar una relación diferente (una relación "can do") entre una interfaz y su tipo de implementación que la herencia. Una interfaz define un subconjunto de funcionalidad (por ejemplo, la posibilidad de probar la igualdad, comparar u ordenar objetos o de admitir análisis y formato con referencia cultural) que la interfaz pone a disposición de sus tipos de implementación.

Tenga en cuenta que "is a" también expresa la relación entre un tipo y una instancia específica de ese tipo. En el ejemplo siguiente, `Automobile` es una clase que tiene tres propiedades de solo lectura exclusivas: `Make`, el

fabricante del automóvil; `Model`, el tipo de automóvil; y `Year`, el año de fabricación. La clase `Automobile` también tiene un constructor cuyos argumentos se asignan a los valores de propiedad, y reemplaza al método Object.ToString para crear una cadena que identifica de forma única la instancia de `Automobile` en lugar de la clase `Automobile`.

```
using System;

public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
           throw new ArgumentNullException("The make cannot be null.");
        else if (String.IsNullOrWhiteSpace(make))
           throw new ArgumentException("make cannot be an empty string or have space characters only.");
        Make = make;

        if (model == null)
           throw new ArgumentNullException("The model cannot be null.");
        else if (String.IsNullOrWhiteSpace(model))
           throw new ArgumentException("model cannot be an empty string or have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
           throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

En este caso, no se debe basar en la herencia para representar marcas y modelos de coche específicos. Por ejemplo, no es necesario definir un tipo `Packard` para representar los automóviles fabricados por la empresa de automóviles Packard Motor. En su lugar, se pueden representar mediante la creación de un objeto `Automobile` con los valores adecuados que se pasan a su constructor de clase, como en el ejemplo siguiente.

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//        1948 Packard Custom Eight
```

Una relación "is a" basada en la herencia se aplica mejor a una clase base y a clases derivadas que agregan miembros adicionales a la clase base o que requieren funcionalidad adicional que no está presente en la clase base.

Diseño de la clase base y las clases derivadas

Veamos el proceso de diseño de una clase base y sus clases derivadas. En esta sección, se definirá una clase base, `Publication`, que representa una publicación de cualquier tipo, como un libro, una revista, un periódico, un diario, un artículo, etc. También se definirá una clase `Book` que se deriva de `Publication`. El ejemplo se podría ampliar fácilmente para definir otras clases derivadas, como `Magazine`, `Journal`, `Newspaper` y `Article`.

Clase base Publication

A la hora de diseñar la clase `Publication`, se deben tomar varias decisiones de diseño:

- Qué miembros se van a incluir en la clase `Publication` base, y si los miembros de `Publication` proporcionan implementaciones de método, o bien si `Publication` es una clase base abstracta que funciona como plantilla para sus clases derivadas.

  En este caso, la clase `Publication` proporcionará implementaciones de método. La sección Diseño de clases base abstractas y sus clases derivadas contiene un ejemplo en el que se usa una clase base abstracta para definir los métodos que deben invalidar las clases derivadas. Las clases derivadas pueden proporcionar cualquier implementación que sea adecuada para el tipo derivado.

  La posibilidad de reutilizar el código (es decir, varias clases derivadas comparten la declaración y la implementación de los métodos de clase base y no tienen que invalidarlos) es una ventaja de las clases base no abstractas. Por tanto, se deben agregar miembros a `Publication` si es probable que algunos o la mayoría de los tipos `Publication` especializados compartan su código. Si no puede proporcionar implementaciones de clase base de forma eficaz, acabará por tener que proporcionar implementaciones de miembros prácticamente idénticas en las clases derivadas en lugar de una única implementación en la clase base. La necesidad de mantener código duplicado en varias ubicaciones es un origen potencial de errores.

  Para maximizar la reutilización del código y crear una jerarquía de herencia lógica e intuitiva, asegúrese de incluir en la clase `Publication` solo los datos y la funcionalidad común a todas o a la mayoría de las publicaciones. Así, las clases derivadas implementan miembros que son únicos para una clase determinada de publicación que representan.

- Hasta qué punto extender la jerarquía de clases. ¿Quiere desarrollar una jerarquía de tres o más clases, en lugar de simplemente una clase base y una o más clases derivadas? Por ejemplo, `Publication` podría ser una clase base de `Periodical`, que, a su vez, es una clase base de `Magazine`, `Journal` y `Newspaper`.

  En el ejemplo, se usará la jerarquía pequeña de una clase `Publication` y una sola clase derivada, `Book`. El ejemplo se podría ampliar fácilmente para crear una serie de clases adicionales que se derivan de `Publication`, como `Magazine` y `Article`.

- Si tiene sentido crear instancias de la clase base. Si no, se debe aplicar la palabra clave abstract a la clase. De lo contrario, se puede crear una instancia de la clase `Publication` mediante una llamada a su constructor de clase. Si se intenta crear una instancia de una clase marcada con la palabra clave `abstract` mediante una llamada directa a su constructor de clase, el compilador de C# genera el error CS0144: "No se puede crear una instancia de la interfaz o clase abstracta". Si se intenta crear una instancia de la clase mediante reflexión, el método de reflexión produce una excepción MemberAccessException.

  De forma predeterminada, se puede crear una instancia de una clase base mediante una llamada a su constructor de clase. No es necesario definir un constructor de clase de forma explícita. Si uno no está presente en el código fuente de la clase base, el compilador de C# proporciona automáticamente un constructor (sin parámetros) de forma predeterminada.

  En el ejemplo, la clase `Publication` se marcará como abstract para que no se puedan crear instancias de ella. Una clase `abstract` sin ningún método `abstract` indica que representa un concepto abstracto que se comparte entre varias clases concretas (como un `Book`, `Journal`).

- Si las clases derivadas deben heredar la implementación de la clase base de determinados miembros, si tienen la opción de invalidar la implementación de la clase base, o bien si deben proporcionar una

implementación. La palabra clave abstract se usa para forzar que las clases derivadas proporcionen una implementación. La palabra clave virtual se usa para permitir que las clases derivadas invaliden un método de clase base. De forma predeterminada, *no se pueden invalidar* los métodos definidos en la clase base.

La clase `Publication` no tiene ningún método `abstract`, pero la propia clase es `abstract`.

- Si una clase derivada representa la clase final en la jerarquía de herencia y no se puede usar ella misma como clase base para clases derivadas adicionales. De forma predeterminada, cualquier clase puede servir como clase base. Se puede aplicar la palabra clave sealed para indicar que una clase no puede servir como clase base para las clases adicionales. Al intentar derivar de una clase sealed se genera el error de compilador CS0509: "no puede derivar del tipo sealed <typeName>".

  Para el ejemplo, la clase derivada se marcará como `sealed`.

En el ejemplo siguiente se muestra el código fuente para la clase `Publication`, así como una enumeración `PublicationType` que devuelve la propiedad `Publication.PublicationType`. Además de los miembros que hereda de Object, la clase `Publication` define los siguientes miembros únicos e invalidaciones de miembros:

```
using System;

public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
   private bool published = false;
   private DateTime datePublished;
   private int totalPages;

   public Publication(string title, string publisher, PublicationType type)
   {
      if (String.IsNullOrWhiteSpace(publisher))
         throw new ArgumentException("The publisher is required.");
      Publisher = publisher;

      if (String.IsNullOrWhiteSpace(title))
         throw new ArgumentException("The title is required.");
      Title = title;

      Type = type;
   }

   public string Publisher { get; }

   public string Title { get; }

   public PublicationType Type { get; }

   public string CopyrightName { get; private set; }

   public int CopyrightDate { get; private set; }

   public int Pages
   {
     get { return totalPages; }
     set
     {
         if (value <= 0)
            throw new ArgumentOutOfRangeException("The number of pages cannot be zero or negative.");
         totalPages = value;
     }
   }

   public string GetPublicationDate()
   {
      if (!published)
```

```
      if (!published)
         return "NYP";
      else
         return datePublished.ToString("d");
   }

   public void Publish(DateTime datePublished)
   {
      published = true;
      this.datePublished = datePublished;
   }

   public void Copyright(string copyrightName, int copyrightDate)
   {
      if (String.IsNullOrWhiteSpace(copyrightName))
         throw new ArgumentException("The name of the copyright holder is required.");
      CopyrightName = copyrightName;

      int currentYear = DateTime.Now.Year;
      if (copyrightDate < currentYear - 10 || copyrightDate > currentYear + 2)
         throw new ArgumentOutOfRangeException($"The copyright year must be between {currentYear - 10} and
{currentYear + 1}");
      CopyrightDate = copyrightDate;
   }

   public override string ToString() => Title;
}
```

- Un constructor

  Dado que la clase `Publication` es `abstract`, no se puede crear una instancia de ella directamente desde código similar al del ejemplo siguiente:

  ```
  var publication = new Publication("Tiddlywinks for Experts", "Fun and Games",
                                    PublicationType.Book);
  ```

  Sin embargo, su constructor de instancia se puede llamar directamente desde los constructores de clases derivadas, como muestra el código fuente de la clase `Book`.

- Dos propiedades relacionadas con la publicación

  `Title` es una propiedad String de solo lectura cuyo valor se suministra mediante la llamada al constructor `Publication`.

  `Pages` es una propiedad Int32 de solo lectura que indica cuántas páginas en total tiene la publicación. El valor se almacena en un campo privado denominado `totalPages`. Debe ser un número positivo o se inicia una excepción ArgumentOutOfRangeException.

- Miembros relacionados con el publicador

  Dos propiedades de solo lectura, `Publisher` y `Type`. Los valores se proporcionan originalmente mediante la llamada al constructor de clase `Publication`.

- Miembros relacionados con la publicación

  Dos métodos, `Publish` y `GetPublicationDate`, establecen y devuelven la fecha de publicación. El método `Publish` establece una marca `published` privada en `true` cuando se llama y asigna la fecha pasada a él como argumento al campo `datePublished` privado. El método `GetPublicationDate` devuelve la cadena "NYP" si la marca `published` es `false`, y el valor del campo `datePublished` si es `true`.

- Miembros relacionados con copyright

El método `Copyright` toma como argumentos el nombre del propietario del copyright y el año del copyright, y los asigna a las propiedades `CopyrightName` y `CopyrightDate`.

- Una invalidación del método `ToString`

  Si un tipo no invalida al método Object.ToString, devuelve el nombre completo del tipo, que es de poca utilidad a la hora de diferenciar una instancia de otra. La clase `Publication` invalida Object.ToString para devolver el valor de la propiedad `Title`.

En la ilustración siguiente se muestra la relación entre la clase base `Publication` y su clase Object heredada de forma implícita.

La clase `Book`.

La clase `Book` representa un libro como un tipo especializado de publicación. En el ejemplo siguiente se muestra el código fuente de la clase `Book`.

```
using System;

public sealed class Book : Publication
{
   public Book(string title, string author, string publisher) :
          this(title, String.Empty, author, publisher)
   { }

   public Book(string title, string isbn, string author, string publisher) : base(title, publisher,
PublicationType.Book)
   {
      // isbn argument must be a 10- or 13-character numeric string without "-" characters.
      // We could also determine whether the ISBN is valid by comparing its checksum digit
      // with a computed checksum.
      //
      if (! String.IsNullOrEmpty(isbn)) {
        // Determine if ISBN length is correct.
        if (! (isbn.Length == 10 | isbn.Length == 13))
            throw new ArgumentException("The ISBN must be a 10- or 13-character numeric string.");
        ulong nISBN = 0;
        if (! UInt64.TryParse(isbn, out nISBN))
            throw new ArgumentException("The ISBN can consist of numeric characters only.");
      }
      ISBN = isbn;

      Author = author;
   }

   public string ISBN { get; }

   public string Author { get; }

   public Decimal Price { get; private set; }

   // A three-digit ISO currency symbol.
   public string Currency { get; private set; }

   // Returns the old price, and sets a new price.
   public Decimal SetPrice(Decimal price, string currency)
   {
       if (price < 0)
          throw new ArgumentOutOfRangeException("The price cannot be negative.");
       Decimal oldValue = Price;
       Price = price;

       if (currency.Length != 3)
          throw new ArgumentException("The ISO currency symbol is a 3-character string.");
       Currency = currency;

       return oldValue;
   }

   public override bool Equals(object obj)
   {
      Book book = obj as Book;
      if (book == null)
         return false;
      else
         return ISBN == book.ISBN;
   }

   public override int GetHashCode() => ISBN.GetHashCode();

   public override string ToString() => $"{(String.IsNullOrEmpty(Author) ? "" : Author + ", ")}{Title}";
}
```

Además de los miembros que hereda de `Publication`, la clase `Book` define los siguientes miembros únicos e invalidaciones de miembros:

- Dos constructores

  Los dos constructores `Book` comparten tres parámetros comunes. Dos, *title* y *publisher*, corresponden a los parámetros del constructor `Publication`. La tercera es *author*, que se almacena para una propiedad `Author` pública inmutable. Un constructor incluye un parámetro *isbn*, que se almacena en la propiedad automática `ISBN`.

  El primer constructor usa esta palabra clave para llamar al otro constructor. El encadenamiento de constructores es un patrón común en la definición de constructores. Los constructores con menos parámetros proporcionan valores predeterminados al llamar al constructor con el mayor número de parámetros.

  El segundo constructor usa la palabra clave base para pasar el título y el nombre del editor al constructor de clase base. Si no realiza una llamada explícita a un constructor de clase base en el código fuente, el compilador de C# proporciona automáticamente una llamada al constructor sin parámetros o predeterminado de la clase base.

- Una propiedad `ISBN` de solo lectura, que devuelve el ISBN (International Standard Book Number) del objeto `Book`, un número exclusivo de 10 y 13 caracteres. El ISBN se proporciona como argumento para uno de los constructores `Book`. El ISBN se almacena en un campo de respaldo privado, generado automáticamente por el compilador.

- Una propiedad `Author` de solo lectura. El nombre del autor se proporciona como argumento para ambos constructores `Book` y se almacena en la propiedad.

- Dos propiedades relacionadas con el precio de solo lectura, `Price` y `Currency`. Sus valores se proporcionan como argumentos en una llamada al método `SetPrice`. La propiedad `Currency` es el símbolo de moneda ISO de tres dígitos (por ejemplo, USD para el dólar estadounidense). Los símbolos de moneda ISO se pueden recuperar de la propiedad ISOCurrencySymbol. Ambas propiedades son de solo lectura desde ubicaciones externas, pero se pueden establecer mediante código en la clase `Book`.

- Un método `SetPrice`, que establece los valores de las propiedades `Price` y `Currency`. Esos son los valores devueltos por dichas propiedades.

- Invalida el método `ToString` (heredado de `Publication`) y los métodos Object.Equals(Object) y GetHashCode (heredados de Object).

  A menos que se invalide, el método Object.Equals(Object) prueba la igualdad de referencia. Es decir, dos variables de objeto se consideran iguales si hacen referencia al mismo objeto. Por otro lado, en la clase `Book`, dos objetos `Book` deben ser iguales si tienen el mismo ISBN.

  Cuando invalide el método Object.Equals(Object), también debe invalidar el método GetHashCode, que devuelve un valor que se usa en el entorno de ejecución para almacenar elementos en colecciones con hash para una recuperación eficiente. El código hash debe devolver un valor que sea coherente con la prueba de igualdad. Puesto que se ha invalidado Object.Equals(Object) para devolver `true`, si las propiedades de ISBN de dos objetos `Book` son iguales, se devuelve el código hash calculado mediante la llamada al método GetHashCode de la cadena devuelta por la propiedad `ISBN`.

En la siguiente ilustración se muestra la relación entre la clase `Book` y `Publication`, su clase base.

Publication
Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

Book
Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, String)
#ctor(String, String, String, String)
PublicationType
Publisher
Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()

Key

Unique member	
Inherited member	
Overridden member	

Ahora se puede crear una instancia de un objeto `Book`, invocar sus miembros únicos y heredados, y pasarla como argumento a un método que espera un parámetro de tipo `Publication` o de tipo `Book`, como se muestra en el ejemplo siguiente.

```
using System;
using static System.Console;

public class Example
{
   public static void Main()
   {
      var book = new Book("The Tempest",  "0971655819", "Shakespeare, William",
                          "Public Domain Press");
      ShowPublicationInfo(book);
      book.Publish(new DateTime(2016, 8, 18));
      ShowPublicationInfo(book);

      var book2 = new Book("The Tempest", "Classic Works Press", "Shakespeare, William");
      Write($"{book.Title} and {book2.Title} are the same publication: " +
            $"{((Publication) book).Equals(book2)}");
   }

   public static void ShowPublicationInfo(Publication pub)
   {
       string pubDate = pub.GetPublicationDate();
       WriteLine($"{pub.Title}, " +
                 $"{(pubDate == "NYP" ? "Not Yet Published" : "published on " + pubDate):d} by
{pub.Publisher}");
   }
}
// The example displays the following output:
//        The Tempest, Not Yet Published by Public Domain Press
//        The Tempest, published on 8/18/2016 by Public Domain Press
//        The Tempest and The Tempest are the same publication: False
```

Diseño de clases base abstractas y sus clases derivadas

En el ejemplo anterior, se define una clase base que proporciona una implementación para una serie de métodos con el fin de permitir que las clases derivadas compartan código. En muchos casos, sin embargo, no se espera que la clase base proporcione una implementación. En su lugar, la clase base es una *clase abstracta* que declara *métodos abstractos*; sirve como una plantilla que define los miembros que debe implementar cada clase derivada. Normalmente, en una clase base abstracta, la implementación de cada tipo derivado es exclusiva de ese tipo. La clase se ha marcado con la palabra clave abstract porque no tenía mucho sentido crear instancias de un objeto `Publication`, aunque la clase proporcionara las implementaciones de funcionalidad común a las publicaciones.

Por ejemplo, cada forma geométrica bidimensional cerrada incluye dos propiedades: área, la extensión interna de la forma; y perímetro, o la distancia a lo largo de los bordes de la forma. La manera en que se calculan estas propiedades, sin embargo, depende completamente de la forma específica. La fórmula para calcular el perímetro (o la circunferencia) de un círculo, por ejemplo, es diferente a la de un triángulo. La clase `Shape` es una clase `abstract` con métodos `abstract`. Eso indica que las clases derivadas comparten la misma funcionalidad, pero que la implementan de otra manera.

En el ejemplo siguiente se define una clase base abstracta denominada `Shape` que define dos propiedades: `Area` y `Perimeter`. Además de marcar la clase con la palabra clave abstract, cada miembro de instancia también se marca con la palabra clave abstract. En este caso, `Shape` también invalida el método Object.ToString para devolver el nombre del tipo, en lugar de su nombre completo. Y define dos miembros estáticos, `GetArea` y `GetPerimeter`, que permiten que los llamadores recuperen fácilmente el área y el perímetro de una instancia de cualquier clase derivada. Cuando se pasa una instancia de una clase derivada a cualquiera de estos métodos, el runtime llama a la invalidación del método de la clase derivada.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }

   public abstract double Perimeter { get; }

   public override string ToString() => GetType().Name;

   public static double GetArea(Shape shape) => shape.Area;

   public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

Después, se pueden derivar algunas clases de `Shape` que representan formas concretas. El ejemplo siguiente define tres clases `Triangle`, `Rectangle` y `Circle`. Cada una usa una fórmula única para esa forma en particular para calcular el área y el perímetro. Algunas de las clases derivadas también definen propiedades, como `Rectangle.Diagonal` y `Circle.Diameter`, que son únicas para la forma que representan.

```
using System;

public class Square : Shape
{
   public Square(double length)
   {
      Side = length;
   }

   public double Side { get; }

   public override double Area => Math.Pow(Side, 2);

   public override double Perimeter => Side * 4;

   public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public double Length { get; }

   public double Width { get; }

   public override double Area => Length * Width;

   public override double Perimeter => 2 * Length + 2 * Width;

   public bool IsSquare() => Length == Width;

   public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) + Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2), 2);

   public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

   // Define a circumference, since it's the more familiar term.
   public double Circumference => Perimeter;

   public double Radius { get; }

   public double Diameter => Radius * 2;
}
```

En el ejemplo siguiente se usan objetos derivados de `Shape`. Se crea una instancia de una matriz de objetos derivados de `Shape` y se llama a los métodos estáticos de la clase `Shape`, que ajusta los valores de propiedad `Shape` devueltos. El runtime recupera los valores de las propiedades invalidadas de los tipos derivados. En el ejemplo también se convierte cada objeto `Shape` de la matriz a su tipo derivado y, si la conversión se realiza correctamente, recupera las propiedades de esa subclase específica de `Shape`.

```
using System;

public class Example
{
   public static void Main()
   {
      Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                         new Circle(3) };
      foreach (var shape in shapes) {
         Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                           $"perimeter, {Shape.GetPerimeter(shape)}");
         var rect = shape as Rectangle;
         if (rect != null) {
            Console.WriteLine($"   Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");
            continue;
         }
         var sq = shape as Square;
         if (sq != null) {
            Console.WriteLine($"   Diagonal: {sq.Diagonal}");
            continue;
         }
      }
   }
}
// The example displays the following output:
//         Rectangle: area, 120; perimeter, 44
//            Is Square: False, Diagonal: 15.62
//         Square: area, 25; perimeter, 20
//            Diagonal: 7.07
//         Circle: area, 28.27; perimeter, 18.85
```

Vea también

- Clases y objetos
- Herencia (Guía de programación de C#)

Uso de Language-Integrated Query (LINQ)

18/03/2020 • 29 minutes to read • Edit Online

Introducción

En este tutorial aprenderá varias características de .NET Core y el lenguaje C#. Aprenderá a:

- Generar secuencias con LINQ.
- Escribir métodos que puedan usarse fácilmente en las consultas LINQ.
- Distinguir entre evaluación diligente y diferida.

Aprenderá estas técnicas mediante la creación de una aplicación que muestra uno de los conocimientos básicos de cualquier mago: el orden aleatorio faro. En resumen, el orden aleatorio faro es una técnica basada en dividir la baraja exactamente por la mitad; a continuación, el orden aleatorio intercala cada carta de cada mitad de la baraja hasta volver a crear la original.

Los magos usan esta técnica porque cada carta está en una ubicación conocida después de cada orden aleatorio, y el orden sigue un patrón de repetición.

Para el propósito sobre el que trata este artículo, resulta divertido ocuparnos de la manipulación de secuencias de datos. La aplicación que se va a crear compilará una baraja de cartas y después realizará una secuencia de órdenes aleatorios, que escribirá cada vez la secuencia completa. También podrá comparar el orden actualizado con el original.

Este tutorial consta de varios pasos. Después de cada paso, puede ejecutar la aplicación y ver el progreso. También puede ver el ejemplo completo en el repositorio dotnet/samples de GitHub. Para obtener instrucciones de descarga, vea Ejemplos y tutoriales.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core. Puede encontrar las instrucciones de instalación en la página Descarga de .NET Core. Puede ejecutar esta aplicación en Windows, Ubuntu Linux, OS X o en un contenedor de Docker. Deberá instalar su editor de código favorito. En las siguientes descripciones se usa Visual Studio Code, que es un editor multiplataforma de código abierto. Sin embargo, puede usar las herramientas que le resulten más cómodas.

Crear la aplicación

El primer paso es crear una nueva aplicación. Abra un símbolo del sistema y cree un nuevo directorio para la aplicación. Conviértalo en el directorio actual. Escriba el comando `dotnet new console` en el símbolo del sistema. Esta acción crea los archivos de inicio para una aplicación básica "Hola a todos".

Si nunca ha usado C# antes, en este tutorial se explica la estructura de un programa con C#. Puede leerlo y después volver aquí para obtener más información sobre LINQ.

Creación del conjunto de datos

Antes de empezar, asegúrese de que las líneas siguientes se encuentran al principio del archivo `Program.cs` generado por `dotnet new console`:

```
// Program.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

Si estas tres líneas (instrucciones `using`) no se encuentran al principio del archivo, nuestro programa no se compilará.

Ahora que tiene todas las referencias que necesitará, tenga en cuenta lo que constituye una baraja de cartas. Habitualmente, una baraja de cartas tiene cuatro palos y cada palo tiene trece valores. Normalmente, podría plantearse crear una clase `Card` directamente del archivo bat y rellenar manualmente una colección de objetos `Card`. Con LINQ, puede ser más conciso que de costumbre al tratar con la creación de una baraja de cartas. En lugar de crear una clase `Card`, puede crear dos secuencias para representar los palos y rangos, respectivamente. Podrá crear un par sencillo de *métodos iterator* que generará las clasificaciones y palos como objetos IEnumerable<T> de cadenas:

```
// Program.cs
// The Main() method

static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Colóquelos debajo del método `Main` en el archivo `Program.cs`. Estos dos métodos utilizan la sintaxis `yield return` para generar una secuencia mientras se ejecutan. El compilador crea un objeto que implementa IEnumerable<T> y genera la secuencia de cadenas conforme se solicitan.

Ahora, puede usar estos métodos iterator para crear la baraja de cartas. Insertará la consulta LINQ en nuestro método `Main`. Aquí tiene una imagen:

```
// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}
```

Las cláusulas múltiples `from` generan una salida SelectMany, que crea una única secuencia a partir de la combinación de cada elemento de la primera secuencia con cada elemento de la segunda secuencia. El orden es importante para nuestros propósitos. El primer elemento de la primera secuencia de origen (palos) se combina con todos los elementos de la segunda secuencia (clasificaciones). Esto genera las trece cartas del primer palo. Dicho proceso se repite con cada elemento de la primera secuencia (palos). El resultado final es una baraja de cartas ordenadas por palos, seguidos de valores.

Es importante tener en cuenta que si decide escribir las instrucciones LINQ en la sintaxis de consulta usada anteriormente o utilizar la sintaxis de método en su lugar, siempre es posible pasar de una forma de sintaxis a la otra. La consulta anterior escrita en la sintaxis de consulta puede escribirse en la sintaxis de método como:

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => new { Suit = suit, Rank = rank }));
```

El compilador convierte las instrucciones LINQ escritas en la sintaxis de consulta a la sintaxis de llamada de método equivalente. Por consiguiente, independientemente de la sintaxis que prefiera, las dos versiones de la consulta producen el mismo resultado. Elija la sintaxis más adecuada a su situación: por ejemplo, si trabaja en un equipo en el que algunos de sus miembros tienen dificultades con la sintaxis de método, procure usar la sintaxis de consulta.

Continúe y ejecute el ejemplo que se ha creado en este punto. Mostrará todas las 52 cartas de la baraja. Puede ser muy útil ejecutar este ejemplo en un depurador para observar cómo se ejecutan los métodos `Suits()` y `Ranks()`. Puede ver claramente que cada cadena de cada secuencia se genera solo según sea necesario.

Manipulación del orden

Seguidamente, céntrese en cómo va a establecer el orden aleatorio de las cartas de la baraja. El primer paso en cualquier orden aleatorio consiste en dividir la baraja en dos. Los métodos Take y Skip que forman parte de las LINQ API le ofrecen esa característica: Colóquelos debajo del bucle `foreach`:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    // 52 cards in a deck, so 52 / 2 = 26
    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
}
```

Pero no existe ningún método de orden aleatorio en la biblioteca estándar que pueda aprovechar, por lo que tendrá que escribir el suyo propio. El método de orden aleatorio que cree mostrará varias técnicas que se utilizan con programas basados en LINQ, por lo que cada parte de este proceso se explica en pasos.

Para agregar alguna funcionalidad a la forma de interactuar con el elemento IEnumerable<T> que obtendrá de las consultas LINQ, tendrá que escribir algunos tipos especiales de métodos llamados métodos de extensión. En resumen, un método de extensión es un *método estático* con una finalidad específica que agrega nueva funcionalidad a un tipo existente sin tener que modificar el tipo original al que quiere agregar la funcionalidad.

Proporcione un nuevo espacio a sus métodos de extensión agregando un nuevo archivo de clase *estática* al programa denominado `Extensions.cs` y comience a compilar el primer método de extensión:

```
// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this IEnumerable<T> first, IEnumerable<T>
second)
        {
            // Your implementation will go here soon enough
        }
    }
}
```

Mire la firma del método por un momento, concretamente los parámetros:

```
public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T> first, IEnumerable<T> second)
```

Puede ver la incorporación del modificador `this` del primer argumento al método. Esto significa que se llama al método como si fuese un método de miembro del tipo del primer argumento. Esta declaración de método también sigue una expresión estándar donde los tipos de entrada y salida son `IEnumerable<T>`. Dicha práctica permite que los métodos LINQ se encadenen entre sí para realizar consultas más complejas.

Naturalmente, dado que dividió la baraja en mitades, tendrá que unir esas mitades. En el código, esto significa que enumerará las dos secuencias adquiridas a través de Take y Skip a la vez, `interleaving` los elementos y crear una sola secuencia: su baraja de cartas recién ordenada aleatoriamente. Escribir un método LINQ que funciona con dos secuencias requiere que comprenda cómo funciona IEnumerable<T>.

La interfaz IEnumerable<T> tiene un método: GetEnumerator. El objeto devuelto por GetEnumerator tiene un método para desplazarse al siguiente elemento, así como una propiedad que recupera el elemento actual de la secuencia. Utilizará estos dos miembros para enumerar la colección y devolver los elementos. Este método de intercalación será un método iterador, por lo que en lugar de crear una colección y devolverla, usará la sintaxis `yield return` anterior.

A continuación se muestra la implementación de ese método:

```
public static IEnumerable<T> InterleaveSequenceWith<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Ahora que ha escrito este método, vuelva al método `Main` y ordene la baraja aleatoriamente una vez:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                       from r in Ranks()
                       select new { Suit = s, Rank = r };

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    var top = startingDeck.Take(26);
    var bottom = startingDeck.Skip(26);
    var shuffle = top.InterleaveSequenceWith(bottom);

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }
}
```

Comparaciones

¿Cuántos órdenes aleatorios se necesitan para devolver la baraja a su orden original? Para averiguarlo, debe escribir un método que determine si dos secuencias son iguales. Cuando ya disponga del método, debe colocar el código que ordena la baraja aleatoriamente en un bucle y comprobarlo para ver cuándo la baraja vuelve a tener su orden original.

Debe ser sencillo escribir un método para determinar si las dos secuencias son iguales. Presenta una estructura similar al método que se escribió para ordenar la baraja aleatoriamente. Solo que esta vez, en lugar de aplicar

`yield return` a cada elemento, se compararán los elementos coincidentes de cada secuencia. Después de que se haya enumerado la secuencia completa, si cada elemento coincide, las secuencias son las mismas:

```
public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

Esto muestra una segunda expresión LINQ: los métodos de terminal. Adoptan una secuencia como entrada (o, en este caso, dos secuencias) y devuelven un único valor escalar. Cuando se utilizan métodos de terminal, siempre son el método final en una cadena de métodos para una consulta LINQ, de ahí el nombre "terminal".

Puede ver esto en acción cuando lo usa para determinar cuándo la baraja vuelve a tener su orden original. Coloque el código de orden aleatorio dentro de un bucle y deténgalo cuando la secuencia vuelva a su orden original, mediante la aplicación del método `SequenceEquals()`. Puede observar que siempre se tratará del método final de cualquier consulta, porque devuelve un único valor en lugar de una secuencia:

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Ejecute el código que tenga hasta el momento y tome nota de cómo la baraja se reorganiza en cada orden aleatorio. Después de ocho órdenes aleatorios (iteraciones del bucle do-while), la baraja vuelve a la configuración original en que se encontraba cuando la creó a partir la consulta LINQ inicial.

Optimizaciones

El ejemplo creado hasta el momento se ejecuta *en orden no aleatorio*, donde las cartas superiores e inferiores son las mismas en cada ejecución. Vamos a realizar un cambio: utilizaremos una ejecución *en orden aleatorio* en su lugar, donde las 52 cartas cambian de posición. Si se trata de un orden aleatorio, intercale la baraja de tal forma que la primera carta de la mitad inferior sea la primera carta de la baraja. Esto significa que la última carta de la mitad superior será la carta inferior. Se trata de un cambio simple en una única línea de código. Actualice la consulta de orden aleatorio actual cambiando las posiciones de Take y Skip. Se cambiará al orden de las mitades superior e inferior de la baraja:

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

Vuelva a ejecutar el programa y verá que, para que la baraja se reordene, se necesitan 52 iteraciones. También empezará a observar algunas degradaciones graves de rendimiento a medida que el programa continúa en ejecución.

Esto se debe a varias razones. Puede que se trate de una de las principales causas de este descenso de rendimiento: un uso ineficaz de la *evaluación diferida*.

En pocas palabras, la evaluación diferida indica que no se realiza la evaluación de una instrucción hasta que su valor es necesario. Las consultas LINQ son instrucciones se evalúan de forma diferida. Las secuencias se generan solo a medida que se solicitan los elementos. Normalmente, es una ventaja importante de LINQ. Sin embargo, en un uso como el que hace este programa, se produce un aumento exponencial del tiempo de ejecución.

Recuerde que la baraja original se generó con una consulta LINQ. Cada orden aleatorio se genera mediante la realización de tres consultas LINQ sobre la baraja anterior. Todas se realizan de forma diferida. Eso también significa que se vuelven a llevar a cabo cada vez que se solicita la secuencia. Cuando llegue a la iteración número 52, habrá regenerado la baraja original demasiadas veces. Se va a escribir un registro para mostrar este comportamiento. A continuación, podrá corregirlo.

En su archivo `Extensions.cs`, escriba o copie el siguiente método. Este método de extensión crea otro archivo denominado `debug.log` en el directorio del proyecto y registra la consulta que se ejecuta actualmente en el archivo de registro. Este método de extensión se puede anexar a una consulta para marcar que se ha ejecutado.

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

Verá un subrayado en zigzag rojo bajo `File`, lo que indica que no existe. No se compilará, ya que el compilador no sabe qué es `File`. Para solucionar este problema, asegúrese de agregar la siguiente línea de código bajo la primera línea de `Extensions.cs`:

```
using System.IO;
```

Esto debería resolver el problema y el error en rojo debería desaparecer.

A continuación, instrumente la definición de cada consulta con un mensaje de registro:

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Rank Generation")
                        select new { Suit = s, Rank = r }).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
                .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
                .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Observe que no se genera un registro cada vez que accede a una consulta. El registro solo se genera cuando crea la consulta original. El programa todavía tarda mucho tiempo en ejecutarse, pero ahora puede ver por qué. Si se le agota la paciencia al ejecutar el orden aleatorio interno con los registros activados, vuelva al orden aleatorio externo. Aún puede ver los efectos de la evaluación diferida. En una ejecución, ejecuta 2592 consultas, incluida toda la generación de palos y valores.

Aquí puede mejorar el rendimiento del código para reducir el número de ejecuciones que realiza. Una corrección sencilla que puede hacer es almacenar en *caché* los resultados de la consulta LINQ original que construye la baraja de cartas. Actualmente, ejecuta las consultas una y otra vez siempre que el bucle do-while pasa por una iteración, lo que vuelve a construir la baraja de cartas y cambia continuamente el orden aleatorio. Para almacenar en caché la baraja de cartas, puede aprovechar los métodos LINQ ToArray y ToList; cuando los anexe a las consultas, realizarán las mismas acciones que les ha indicado que hagan, pero ahora almacenarán los resultados en una matriz o una lista, según el método al que elija llamar. Anexe el método ToArray de LINQ a las dos consultas y ejecute de nuevo el programa:

```
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                        from r in Ranks().LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom Half"))
            .LogQuery("Shuffle")
            .ToArray();
        */

        shuffle = shuffle.Skip(26)
            .LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle")
            .ToArray();

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

Ahora el orden aleatorio externo se reduce a 30 consultas. Vuelva a ejecutarlo con el orden aleatorio interno y verá mejoras similares: ahora ejecuta 162 consultas.

Este ejemplo está **diseñado** para resaltar los casos de uso en que la evaluación diferida puede generar dificultades de rendimiento. Si bien es importante ver dónde la evaluación diferida puede afectar al rendimiento del código, es igualmente importante entender que no todas las consultas deben ejecutarse de manera diligente. El resultado de rendimiento en el que incurre sin usar ToArray se debe a que cada nueva disposición de la baraja de cartas se crea a partir de la disposición anterior. La evaluación diferida supone que cada nueva configuración de la baraja se realiza a partir de la baraja original, incluso con la ejecución del código que crea el elemento `startingDeck`. Esto conlleva una gran cantidad de trabajo adicional.

En la práctica, algunos algoritmos se ejecutan bien con la evaluación diligente y otros, con la evaluación diferida. Para el uso diario, la evaluación diferida suele ser una mejor opción cuando el origen de datos es un proceso independiente, como un motor de base de datos. Para las bases de datos, la evaluación diferida permite realizar consultas más complejas que ejecuten un solo recorrido de ida y vuelta al procesamiento de la base de datos y vuelvan al resto del código. LINQ es flexible tanto si decide usar la evaluación diligente como la diferida, así que

calibre sus procesos y elija el tipo de evaluación que le ofrece el mejor rendimiento.

Conclusión

En este proyecto ha tratado lo siguiente:

- Uso de consultas LINQ para agregar datos a una secuencia significativa
- Escritura de métodos de extensión para agregar nuestra propia funcionalidad personalizada a las consultas LINQ
- Localización de áreas en nuestro código donde nuestras consultas LINQ pueden tener problemas de rendimiento como la degradación de la velocidad
- Evaluación diligente y diferida en lo que respecta a las consultas LINQ y las implicaciones que podrían tener en el rendimiento de la consulta

Aparte de LINQ, ha aprendido algo sobre una técnica que los magos utilizan para hacer trucos de cartas. Los magos usan el orden aleatorio Faro porque les permite controlar dónde está cada carta en la baraja. Ahora que lo conoce, no se lo estropee a los demás.

Para más información sobre LINQ, vea:

- Language-Integrated Query (LINQ)
- Introducción a LINQ
- Operaciones básicas de consulta LINQ (C#)
- Transformaciones de datos con LINQ (C#)
- Sintaxis de consultas y sintaxis de métodos en LINQ (C#)
- Características de C# compatibles con LINQ

Uso de atributos en C#

14/04/2020 • 14 minutes to read • Edit Online

Los atributos proporcionan una manera de asociar la información con el código de manera declarativa. También pueden proporcionar un elemento reutilizable que se puede aplicar a diversos destinos.

Considere el atributo `[Obsolete]`. Se puede aplicar a clases, structs, métodos, constructores y más. *Declara* que el elemento está obsoleto. Es decisión del compilador de C# buscar este atributo y realizar alguna acción como respuesta.

En este tutorial, se le introducirá a cómo agregar atributos al código, cómo crear y usar sus propios atributos y cómo usar algunos atributos que se integran en .NET Core.

Requisitos previos

Deberá configurar la máquina para ejecutar .NET Core. Puede encontrar las instrucciones de instalación en la página Descargas de .NET Core. Puede ejecutar esta aplicación en Windows, Ubuntu Linux, macOS o en un contenedor de Docker. Deberá instalar su editor de código favorito. En las siguientes descripciones se usa Visual Studio Code, que es un editor multiplataforma de código abierto. Sin embargo, puede usar las herramientas que le resulten más cómodas.

Crear la aplicación

Ahora que ha instalado todas las herramientas, cree una nueva aplicación de .NET Core. Para usar el generador de línea de comandos, ejecute el siguiente comando en su shell favorito:

```
dotnet new console
```

Este comando creará archivos de proyecto esenciales de .NET Core. Debe ejecutar `dotnet restore` para restaurar las dependencias necesarias para compilar este proyecto.

No es necesario ejecutar `dotnet restore` porque lo ejecutan implícitamente todos los comandos que necesitan que se produzca una restauración, como `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` y `dotnet pack`. Para deshabilitar la restauración implícita, use la opción `--no-restore`.

El comando `dotnet restore` sigue siendo válido en algunos escenarios donde tiene sentido realizar una restauración explícita, como las compilaciones de integración continua en Azure DevOps Services o en los sistemas de compilación que necesitan controlar explícitamente cuándo se produce la restauración.

Para obtener información sobre cómo administrar fuentes de NuGet, vea la documentación de `dotnet restore`.

Para ejecutar el programa, use `dotnet run`. Deberá ver la salida "Hola a todos" a la consola.

Cómo agregar atributos al código

En C#, los atributos son clases que se heredan de la clase base `Attribute`. Cualquier clase que se hereda de `Attribute` puede usarse como una especie de "etiqueta" en otros fragmentos de código. Por ejemplo, hay un atributo llamado `ObsoleteAttribute`, que se usa para indicar que el código está obsoleto y ya no debe utilizarse. Puede colocar este atributo en una clase, por ejemplo, mediante corchetes.

```
[Obsolete]
public class MyClass
{
}
```

Tenga en cuenta que, mientras que la clase se denomina `ObsoleteAttribute`, solo es necesario usar `[Obsolete]` en el código. Se trata de una convención que sigue C#. Puede usar el nombre completo `[ObsoleteAttribute]` si así lo prefiere.

Cuando se marca una clase como obsoleta, es una buena idea proporcionar alguna información de *por qué* es obsoleta, o *qué* usar en su lugar. Para ello se pasa un parámetro de cadena al atributo obsoleto.

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

La cadena se pasa como argumento a un constructor `ObsoleteAttribute`, como si estuviera escribiendo `var attr = new ObsoleteAttribute("some string")`.

Los parámetros a un constructor de atributos se limitan a literales o tipos simples: `bool, int, double, string, Type, enums, etc` y matrices de esos tipos. No se puede usar una expresión o una variable. Es libre de usar parámetros posicionales o con nombre.

Cómo crear su propio atributo

Crear un atributo es tan sencillo como heredarlo de la clase base `Attribute`.

```
public class MySpecialAttribute : Attribute
{
}
```

Con lo anterior, ahora puedo usar `[MySpecial]` (o `[MySpecialAttribute]`) como un atributo en otra parte de la base de código.

```
[MySpecial]
public class SomeOtherClass
{
}
```

Los atributos de la biblioteca de clases base de .NET como `ObsoleteAttribute` desencadenan ciertos comportamientos en el compilador. Sin embargo, cualquier atributo que cree funcionará como metadatos y no tendrá como resultado ningún código dentro de la clase de atributo que se ejecuta. Es decisión suya actuar sobre esos metadatos en otra parte del código (más adelante en este tutorial se hablará sobre ello).

Aquí hay un problema que se debe vigilar. Como se mencionó anteriormente, solo determinados tipos se pueden pasar como argumentos al usar atributos. Sin embargo, al crear un tipo de atributo, el compilador de C# no le impedirá crear esos parámetros. En el ejemplo siguiente, he creado un atributo con un constructor que se compila correctamente.

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str) {
    }
}
```

Sin embargo, no podrá usar este constructor con una sintaxis de atributo.

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

Lo anterior provocará un error de compilador como `Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`.

Cómo restringir el uso de atributos

Los atributos pueden usarse en varios destinos. Los ejemplos anteriores los muestran en las clases, pero ahora se pueden usar en:

- Ensamblado
- Clase
- Constructor
- delegado
- Enum
- evento
- Campo
- GenericParameter
- Interfaz
- Método
- Module
- Parámetro
- Propiedad.
- ReturnValue
- Struct

Cuando se crea una clase de atributo, de forma predeterminada C# podrá usar ese atributo en cualquiera de los destinos de atributo posibles. Si desea restringir el atributo a determinados destinos, puede hacerlo mediante la clase de atributo `AttributeUsageAttribute`. ¡Sí, un atributo en un atributo!

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

Si intenta colocar el atributo anterior en algo que no sea una clase o un struct, obtendrá un error del compilador como

```
Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations
```

.

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

Cómo usar los atributos asociados a un elemento de código

Los atributos actúan como metadatos. Sin algo de fuerza exterior, no harían realmente nada.

Para buscar y actuar sobre los atributos, se requiere en general reflexión. En este tutorial no se tratará la reflexión de forma detallada, pero la idea básica es que la reflexión le permite escribir código en C# que examina otro código.

Por ejemplo, puede usar reflexión para obtener información sobre una clase (agregue `using System.Reflection;` al principio del código):

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

La salida será algo como esto:

```
The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null
```

.

Una vez que tenga un objeto `TypeInfo` (o un elemento `MemberInfo`, `FieldInfo`, etc.), puede usar el método `GetCustomAttributes`. Este devolverá una colección de objetos `Attribute`. También puede usar `GetCustomAttribute` y especificar un tipo de atributo.

Este es un ejemplo del uso de `GetCustomAttributes` en una instancia de `MemberInfo` para `MyClass` (que anteriormente vimos que contiene un atributo `[Obsolete]`).

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

Eso se imprimirá en la consola: `Attribute on MyClass: ObsoleteAttribute`. Intente agregar otros atributos a `MyClass`.

Es importante tener en cuenta que se crean instancias de estos objetos `Attribute` de forma diferida. Es decir, no podrá crea una instancia de ellos hasta que use `GetCustomAttribute` o `GetCustomAttributes`. También se crea una instancia de ellos cada vez. Al llamar a `GetCustomAttributes` dos veces en una fila se devuelven dos instancias diferentes de `ObsoleteAttribute`.

Atributos comunes en la biblioteca de clases base (BCL)

Muchas herramientas y marcos de trabajo emplean atributos. NUnit usa atributos como `[Test]` y `[TestFixture]` que son utilizados por la serie de pruebas NUnit. ASP.NET MVC usa atributos como `[Authorize]` y proporciona un marco de filtro de acción para ejecutar problemas transversales en acciones de MVC. PostSharp usa la sintaxis de atributo para permitir la programación en C# orientada a aspectos.

Estos son algunos atributos importantes integrados en las bibliotecas de clases base de .NET Core:

- `[Obsolete]`. Este se usó en los ejemplos anteriores, y se encuentra en el espacio de nombres `System`. Es útil proporcionar documentación declarativa sobre una base de código cambiante. Se puede proporcionar un mensaje en forma de cadena, y se puede usar otro parámetro booleano para escalarlo de una advertencia del compilador a un error del compilador.
- `[Conditional]`. Este atributo está en el espacio de nombres `System.Diagnostics`. Este atributo se puede aplicar a métodos (o clases de atributos). Debe pasar una cadena al constructor. Si esa cadena no coincide con una directiva `#define`, el compilador de C# quitará las llamadas a ese método (pero no el método propiamente dicho). Normalmente se usa con fines de depuración (diagnósticos).
- `[CallerMemberName]`. Este atributo se puede usar en parámetros, y reside en el espacio de nombres `System.Runtime.CompilerServices`. Es un atributo que se usa para inyectar el nombre del método que llama a otro método. Esto se usa normalmente como una manera de eliminar "cadenas mágicas" al implementar INotifyPropertyChanged en distintos marcos de interfaz de usuario. Por ejemplo:

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    private string _name;
    public string Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not needed here explicitly
            }
        }
    }
}
```

En el código anterior, no necesita tener una cadena `"Name"` de literal. Esto puede ayudar a impedir errores relacionados con los tipos y también agiliza los procesos de refactorización o cambio de nombre.

Resumen

Los atributos traen la eficacia declarativa a C#, pero son una forma de metadatos de código y no actúan por sí mismos.

Paseo por el lenguaje C#

25/04/2020 • 10 minutes to read • Edit Online

C# (pronunciado "si sharp" en inglés) es un lenguaje de programación moderno, basado en objetos y con seguridad de tipos. C# tiene sus raíces en la familia de lenguajes C, y a los programadores de C, C++, Java y JavaScript les resultará familiar inmediatamente.

Este paseo proporciona información general de los principales componentes del lenguaje en C# 8 y versiones anteriores. Si quiere explorar el lenguaje a través de ejemplos interactivos, pruebe los tutoriales de introducción a C#.

C# es un lenguaje orientado a objetos, pero también incluye compatibilidad para programación ***orientada a componentes***. El diseño de software contemporáneo se basa cada vez más en componentes de software en forma de paquetes independientes y autodescriptivos de funcionalidad. El aspecto más importante de estos componentes es que presentan un modelo de programación con propiedades, métodos y eventos. Tienen atributos que proporcionan información declarativa sobre el componente. Además, incorporan su propia documentación. C# proporciona construcciones de lenguaje para admitir directamente estos conceptos, por lo que se trata de un lenguaje natural en el que crear y usar componentes de software.

Varias características de C# ayudan en la creación de aplicaciones sólidas y duraderas. ***La recolección de elementos no utilizados*** permite recuperar automáticamente la memoria ocupada por objetos no utilizados inalcanzables. Con ***el control de excepciones*** se proporciona un enfoque estructurado y extensible para la detección y recuperación de errores. El ***diseño con seguridad de tipos*** de este lenguaje hace que sea imposible leer las variables sin inicializar, indizar las matrices más allá de sus límites o realizar conversiones de tipos no comprobados.

C# tiene un ***sistema de tipo unificado***. Todos los tipos de C#, incluidos los tipos primitivos como `int` y `double`, se heredan de un único tipo `object` raíz. Por lo tanto, todos los tipos comparten un conjunto de operaciones comunes, y los valores de todos los tipos se pueden almacenar, transportar y utilizar de manera coherente. Además, C# admite tipos de valor y tipos de referencia definidos por el usuario, lo que permite la asignación dinámica de objetos, así como almacenamiento en línea de estructuras ligeras.

Para asegurarse de que las programas y las bibliotecas de C# pueden evolucionar a lo largo del tiempo de manera compatible, se ha puesto mucho énfasis en el ***versionamiento*** del diseño de C#. Muchos lenguajes de programación prestan poca atención a este problema y, como resultado, los programas escritos en dichos lenguajes se interrumpen con más frecuencia de lo necesario cuando se introducen nuevas versiones de las bibliotecas dependientes. Los aspectos del diseño de C# afectados directamente por las consideraciones de versionamiento incluyen los modificadores `virtual` y `override` independientes, las reglas para la resolución de sobrecargas de métodos y la compatibilidad para declaraciones explícitas de miembros de interfaz.

En versiones más recientes, C# ha adoptado otros paradigmas de programación. C# ha incluido características que admiten técnicas de programación funcional, como las expresiones lambda. Otras características nuevas admiten la separación de datos y algoritmos, como la coincidencia de patrones.

Hola a todos

El programa "Hola mundo" tradicionalmente se usa para presentar un lenguaje de programación. En este caso, se usa C#:

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

Normalmente, los archivos de código fuente de C# tienen la extensión de archivo `.cs`. Para crear este programa, primero descargue e instale el SDK de .NET Core. Después, ejecute el comando `dotnet new console -o hello` para crear un programa y un script de compilación. El programa y el script de compilación se encuentran en los archivos `Program.cs` y `hello.csproj` respectivamente. Compilará y ejecutará la aplicación con los comandos `run`:

```
cd hello
dotnet run
```

El programa produce el siguiente resultado:

```
Hello, World!
```

El programa "Hola mundo" empieza con una directiva `using` que hace referencia al espacio de nombres `System`. Los espacios de nombres proporcionan un método jerárquico para organizar las bibliotecas y los programas de C#. Los espacios de nombres contienen tipos y otros espacios de nombres; por ejemplo, el espacio de nombres `System` contiene varios tipos, como la clase `Console` a la que se hace referencia en el programa, y otros espacios de nombres, como `IO` y `Collections`. Una directiva `using` que hace referencia a un espacio de nombres determinado permite el uso no calificado de los tipos que son miembros de ese espacio de nombres. Debido a la directiva `using`, puede utilizar el programa `Console.WriteLine` como abreviatura de `System.Console.WriteLine`.

La clase `Hello` declarada por el programa "Hola mundo" tiene un miembro único, el método llamado `Main`. El método `Main` se declara con el modificador static. Mientras que los métodos de instancia pueden hacer referencia a una instancia de objeto envolvente determinada utilizando la palabra clave `this`, los métodos estáticos funcionan sin referencia a un objeto determinado. Por convención, un método estático denominado `Main` sirve como punto de entrada de un programa.

La salida del programa la genera el método `WriteLine` de la clase `Console` en el espacio de nombres `System`. Esta clase la proporcionan las bibliotecas de clase estándar, a las que, de forma predeterminada, el compilador hace referencia automáticamente.

Elementos del lenguaje C#

Hay mucha más información sobre C#. Los temas siguientes proporcionan introducciones a los elementos del lenguaje C#. Estas introducciones proporcionan información básica sobre todos los elementos del lenguaje y ofrecen los detalles necesarios para profundizar más:

- Estructura del programa
  - Conozca los principales conceptos organizativos del lenguaje C#: ***programas***, ***espacios de nombres***, ***tipos***, ***miembros*** y ***ensamblados***.
- Tipos y variables
  - Obtenga información sobre los ***tipos de valor***, los ***tipos de referencia*** y las ***variables*** del lenguaje C#.
- Expresiones

- Las ***expresiones*** se construyen con ***operandos*** y ***operadores***. Las expresiones producen un valor.
- Instrucciones
  - Use ***instrucciones*** para expresar las acciones de un programa.
- Clases y objetos
  - Las ***clases*** son los tipos más fundamentales de C#. Los ***objetos*** son instancias de una clase. Las clases se generan mediante ***miembros***, que también se tratan en este tema.
- Matrices
  - Una ***matriz*** es una estructura de datos que contiene un número de variables a las que se accede mediante índices calculados.
- Interfaces
  - Una ***interfaz*** define un contrato que se puede implementar mediante clases y structs. Una interfaz puede contener métodos, propiedades, eventos e indexadores. Una interfaz no proporciona implementaciones de los miembros que define, simplemente especifica los miembros que se deben proporcionar mediante clases o structs que implementan la interfaz.
- Delegados
  - Un ***tipo de delegado*** representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son similares al concepto de punteros de función en otros lenguajes, pero a diferencia de los punteros de función, los delegados están orientados a objetos y presentan seguridad de tipos.
- Atributos
  - Los ***atributos*** permiten a los programas especificar información declarativa adicional sobre los tipos, miembros y otras entidades.

> **NOTE**
> Estos artículos se aplican a C# 7.0 y versiones posteriores. Es posible que algunas características no estén disponibles en versiones anteriores.

SIGUIENTE

Estructura del programa

27/04/2020 • 5 minutes to read • Edit Online

Los principales conceptos organizativos en C# son ***programas***, ***espacios de nombres***, ***tipos***, ***miembros*** y ***ensamblados***. Los programas de C# constan de uno o más archivos de origen. Los programas declaran tipos, que contienen miembros y pueden organizarse en espacios de nombres. Las clases e interfaces son ejemplos de tipos. Los campos, los métodos, las propiedades y los eventos son ejemplos de miembros. Cuando se compilan programas de C#, se empaquetan físicamente en ensamblados. Normalmente, los ensamblados tienen la extensión de archivo `.exe` o `.dll`, dependiendo de si implementan ***aplicaciones*** o ***bibliotecas***, respectivamente.

Puede crear un proyecto de biblioteca denominado *acme* mediante el comando `dotnet new`:

```
dotnet new classlib -o acme
```

En ese proyecto, declare una clase denominada `Stack` en un espacio de nombres llamado `Acme.Collections`:

```
using System;
namespace Acme.Collections
{
    public class Stack
    {
        Entry top;
        public void Push(object data)
        {
            top = new Entry(top, data);
        }

        public object Pop()
        {
            if (top == null)
            {
                throw new InvalidOperationException();
            }
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;
            public Entry(Entry next, object data)
            {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

El nombre completo de esta clase es `Acme.Collections.Stack`. La clase contiene varios miembros: un campo denominado `top`, dos métodos denominados `Push` y `Pop`, y una clase anidada denominada `Entry`. La clase `Entry` contiene además tres miembros: un campo denominado `next`, un campo denominado `data` y un constructor. El comando:

```
dotnet build
```

compila el ejemplo como una biblioteca (código sin un punto de entrada `Main` y genera un ensamblado denominado `acme.dll`.

Los ensamblados contienen código ejecutable en forma de instrucciones de lenguaje intermedio (IL) e información simbólica en forma de metadatos. Antes de ejecutarlo, el compilador Just-In-Time (JIT) del entorno de ejecución de .NET convierte el código de IL de un ensamblado en código específico del procesador.

Como un ensamblado es una unidad autodescriptiva de funcionalidad que contiene código y metadatos, no hay necesidad de directivas `#include` ni archivos de encabezado de C#. Los tipos y miembros públicos contenidos en un ensamblado determinado estarán disponibles en un programa de C# simplemente haciendo referencia a dicho ensamblado al compilar el programa. Por ejemplo, este programa usa la clase `Acme.Collections.Stack` desde el ensamblado `acme.dll`:

```
using System;
using Acme.Collections;
class Example
{
    static void Main()
    {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

El archivo *csproj* del proyecto del programa anterior debe incluir un nodo de referencia para que el compilador de C# resuelva las referencias a las clases del ensamblado `acme.dll`:

```
  <ItemGroup>
    <ProjectReference Include="..\acme\acme.csproj" />
  </ItemGroup>
```

Después de agregarlo, `dotnet build` crea un ensamblado ejecutable denominado `example.exe` que, cuando se ejecuta, produce esta salida:

```
100
10
1
```

C# permite que el texto de origen de un programa se almacene en varios archivos de origen. Cuando se compila un programa de C# de varios archivos, todos los archivos de origen se procesan juntos y los archivos de origen pueden hacerse referencia mutuamente de forma libre; desde un punto de vista conceptual, es como si todos los archivos de origen estuvieran concatenados en un archivo de gran tamaño antes de ser procesados. En C# nunca se necesitan declaraciones adelantadas porque, excepto en contadas ocasiones, el orden de declaración es insignificante. C# no limita un archivo de origen a declarar solamente un tipo público ni precisa que el nombre del archivo de origen coincida con un tipo declarado en el archivo de origen.

Tipos y variables

04/05/2020 • 12 minutes to read • Edit Online

Hay dos clases de tipos en C#: *tipos de valor* y *tipos de referencia*. Las variables de tipos de valor contienen directamente los datos, mientras que las variables de los tipos de referencia almacenan referencias a los datos, lo que se conoce como objetos. Con los tipos de referencia, es posible que dos variables hagan referencia al mismo objeto y que, por tanto, las operaciones en una variable afecten al objeto al que hace referencia la otra. Con los tipos de valor, cada variable tiene su propia copia de los datos y no es posible que las operaciones en una variable afecten a la otra (excepto para las variables de parámetro `ref` y `out`).

Los tipos de valor de C# se dividen en *tipos simples*, *tipos de enumeración*, *tipos de estructura* y *tipos de valor que aceptan valores NULL*. Los tipos de referencia de C# se dividen en *tipos de clase*, *tipos de interfaz*, *tipos de matriz* y *tipos delegados*.

En el esquema siguiente se ofrece información general del sistema de tipos de C#.

- Tipos de valor
  - Tipos simples
    - Entero con signo: `sbyte`, `short`, `int`, `long`
    - Entero sin signo: `byte`, `ushort`, `uint`, `ulong`
    - Caracteres Unicode: `char`
    - Punto flotante binario IEEE: `float`, `double`
    - Punto flotante decimal de alta precisión: `decimal`
    - Booleano: `bool`
  - Tipos de enumeración
    - Tipos definidos por el usuario con el formato `enum E {...}`
  - Tipos de estructura
    - Tipos definidos por el usuario con el formato `struct S {...}`
  - Tipos de valores que aceptan valores NULL
    - Extensiones de todos los demás tipos de valor con un valor `null`
  - Tipos de valor de tupla
    - Tipos definidos por el usuario con el formato `(T1, T2, ...)`
- Tipos de referencia
  - Tipos de clase
    - Clase base definitiva de todos los demás tipos: `object`
    - Cadenas Unicode: `string`
    - Tipos definidos por el usuario con el formato `class C {...}`
  - Tipos de interfaz
    - Tipos definidos por el usuario con el formato `interface I {...}`
  - Tipos de matriz
    - Unidimensional y multidimensional; por ejemplo, `int[]` y `int[,]`
  - Tipos delegados
    - Tipos definidos por el usuario con el formato `delegate int D(...)`

Para obtener más información sobre los tipos numéricos, vea Tipos enteros y Tabla de tipos de punto flotante.

El tipo `bool` de C# se utiliza para representar valores booleanos, valores que son `true` o `false`.

El procesamiento de caracteres y cadenas en C# utiliza la codificación Unicode. El tipo `char` representa una unidad de código UTF-16 y el tipo `string` representa una secuencia de unidades de código UTF-16.

Los programas de C# utilizan *declaraciones de tipos* para crear nuevos tipos. Una declaración de tipos especifica el nombre y los miembros del nuevo tipo. Cinco de las categorías de tipos de C# las define el usuario: tipos de clase, tipos de estructura, tipos de interfaz, tipos de enumeración y tipos delegados.

A tipo `class` define una estructura de datos que contiene miembros de datos (campos) y miembros de función (métodos, propiedades y otros). Los tipos de clase admiten herencia única y polimorfismo, mecanismos por los que las clases derivadas pueden extender y especializar clases base.

Un tipo `struct` es similar a un tipo de clase, por el hecho de que representa una estructura con miembros de datos y miembros de función. Pero a diferencia de las clases, las estructuras son tipos de valor y no suelen requerir la asignación del montón. Los tipos de estructura no admiten la herencia especificada por el usuario y todos se heredan implícitamente del tipo `object`.

Un tipo `interface` define un contrato como un conjunto con nombre de miembros de función públicos. Un valor `class` o `struct` que implementa `interface` debe proporcionar implementaciones de miembros de función de la interfaz. Un `interface` puede heredar de varias interfaces base, y un `class` o `struct` pueden implementar varias interfaces.

Un tipo `delegate` representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son análogos a los tipos de función proporcionados por los lenguajes funcionales. También son similares al concepto de punteros de función de otros lenguajes. A diferencia de los punteros de función, los delegados están orientados a objetos y tienen seguridad de tipos.

Los tipos `class`, `struct`, `interface` y `delegate` admiten parámetros genéricos, mediante los que se pueden parametrizar con otros tipos.

Un tipo `enum` es un tipo distinto con constantes con nombre. Cada tipo `enum` tiene un tipo subyacente, que debe ser uno de los ocho tipos enteros. El conjunto de valores de un tipo `enum` es igual que el conjunto de valores del tipo subyacente.

C# admite matrices unidimensionales y multidimensionales de cualquier tipo. A diferencia de los tipos enumerados antes, no es necesario declarar los tipos de matriz antes de usarlos. En su lugar, los tipos de matriz se crean mediante un nombre de tipo entre corchetes. Por ejemplo, `int[]` es una matriz unidimensional de `int`, `int[,]` es una matriz bidimensional de `int` y `int[][]` es una matriz unidimensional de la matriz unidimensional de `int`.

Tampoco es necesario declarar los tipos que admiten un valor NULL antes de usarlos. Para cada tipo de valor `T` que no acepta valores NULL, existe un tipo de valor `T?` que admite un valor NULL correspondiente, que puede tener un valor adicional, `null`. Por ejemplo, `int?` es un tipo que puede contener cualquier número entero de 32 bits o el valor `null`.

El sistema de tipos de C# está unificado, de tal forma que un valor de cualquier tipo puede tratarse como `object`. Todos los tipos de C# directa o indirectamente se derivan del tipo de clase `object`, y `object` es la clase base definitiva de todos los tipos. Los valores de tipos de referencia se tratan como objetos mediante la visualización de los valores como tipo `object`. Los valores de tipos de valor se tratan como objetos mediante la realización de *operaciones de conversión boxing* y *operaciones de conversión unboxing*. En el ejemplo siguiente, un valor `int` se convierte en `object` y vuelve a `int`.

```
using System;
class BoxingExample
{
    static void Main()
    {
        int i = 123;
        object o = i;    // Boxing
        int j = (int)o;  // Unboxing
    }
}
```

Cuando se asigna un valor de un tipo de valor a una referencia `object`, se asigna un "box" para contener el valor. Ese box es una instancia de un tipo de referencia, y es donde se copia el valor. Por el contrario, cuando una referencia `object` se convierte en un tipo de valor, se comprueba si el elemento `object` al que se hace referencia es un box del tipo de valor correcto. Si la comprobación se realiza correctamente, el valor del box se copia en el tipo de valor.

El sistema de tipos unificado de C# conlleva efectivamente que los tipos de valor se tratan como referencias `object` "a petición". Debido a la unificación, las bibliotecas de uso general que utilizan el tipo `object` pueden usarse con todos los tipos que se derivan de `object`, como, por ejemplo, los tipos de referencia y los tipos de valor.

Hay varios tipos de *variables* en C#, entre otras, campos, elementos de matriz, variables locales y parámetros. Las variables representan ubicaciones de almacenamiento, y cada variable tiene un tipo que determina qué valores pueden almacenarse en la variable, como se muestra a continuación.

- Tipo de valor distinto a NULL
  - Un valor de ese tipo exacto
- Tipos de valor NULL
  - Un valor `null` o un valor de ese tipo exacto
- objeto
  - Una referencia `null`, una referencia a un objeto de cualquier tipo de referencia o una referencia a un valor de conversión boxing de cualquier tipo de valor
- Tipo de clase
  - Una referencia `null`, una referencia a una instancia de ese tipo de clase o una referencia a una instancia de una clase derivada de ese tipo de clase
- Tipo de interfaz
  - Un referencia `null`, una referencia a una instancia de un tipo de clase que implementa dicho tipo de interfaz o una referencia a un valor de conversión boxing de un tipo de valor que implementa dicho tipo de interfaz
- Tipo de matriz
  - Una referencia `null`, una referencia a una instancia de ese tipo de matriz o una referencia a una instancia de un tipo de matriz compatible
- Tipo delegado
  - Una referencia `null` o una referencia a una instancia de un tipo delegado compatible

ANTERIOR SIGUIENTE

Expresiones

18/03/2020 • 3 minutes to read • Edit Online

Las *expresiones* se construyen con *operandos* y *operadores*. Los operadores de una expresión indican qué operaciones se aplican a los operandos. Ejemplos de operadores incluyen `+`, `-`, `*`, `/` y `new`. Algunos ejemplos de operandos son literales, campos, variables locales y expresiones.

Cuando una expresión contiene varios operadores, la *precedencia* de los operadores controla el orden en que se evalúan los operadores individuales. Por ejemplo, la expresión `x + y * z` se evalúa como `x + (y * z)` porque el operador `*` tiene mayor precedencia que el operador `+`.

Cuando un operando se encuentra entre dos operadores con la misma precedencia, la *asociatividad* de los operadores controla el orden en que se realizan las operaciones:

- Excepto los operadores de asignación y los operadores de fusión de NULL, todos los operadores binarios son *asociativos a la izquierda*, lo que significa que las operaciones se realizan de izquierda a derecha. Por ejemplo, `x + y + z` se evalúa como `(x + y) + z`.
- Los operadores de asignación, los operadores de fusión de NULL `??` y `??=` y el operador condicional `?:` son *asociativos a la derecha*, lo que significa que las operaciones se realizan de derecha a izquierda. Por ejemplo, `x = y = z` se evalúa como `x = (y = z)`.

La precedencia y la asociatividad pueden controlarse mediante paréntesis. Por ejemplo, `x + y * z` primero multiplica `y` por `z` y luego suma el resultado a `x`, pero `(x + y) * z` primero suma `x` y `y` y luego multiplica el resultado por `z`.

La mayoría de los operadores se pueden *sobrecargar*. La sobrecarga de operador permite la especificación de implementaciones de operadores definidas por el usuario para operaciones donde uno o ambos operandos son de un tipo de struct o una clase definidos por el usuario.

C# ofrece una serie de operadores para realizar operaciones aritméticas, lógicas, de desplazamiento y bit a bit, además de comparaciones de igualdad y de orden.

Para obtener la lista completa de los operadores de C# ordenados por nivel de prioridad, vea Operadores de C#.

ANTERIOR SIGUIENTE

Instrucciones

18/03/2020 • 5 minutes to read • Edit Online

Las acciones de un programa se expresan mediante *instrucciones*. C# admite varios tipos de instrucciones diferentes, varias de las cuales se definen en términos de instrucciones insertadas.

Un *bloque* permite que se escriban varias instrucciones en contextos donde se permite una única instrucción. Un bloque se compone de una lista de instrucciones escritas entre los delimitadores `{` y `}`.

Las *instrucciones de declaración* se usan para declarar variables locales y constantes.

Las *instrucciones de expresión* se usan para evaluar expresiones. Las expresiones que pueden usarse como instrucciones incluyen invocaciones de método, asignaciones de objetos mediante el operador `new`, asignaciones mediante `=` y los operadores de asignación compuestos, operaciones de incremento y decremento mediante los operadores `++` y `--` y expresiones `await`.

Las *instrucciones de selección* se usan para seleccionar una de varias instrucciones posibles para su ejecución en función del valor de alguna expresión. Este grupo contiene las instrucciones `if` y `switch`.

Las *instrucciones de iteración* se usan para ejecutar una instrucción insertada de forma repetida. Este grupo contiene las instrucciones `while`, `do`, `for` y `foreach`.

Las *instrucciones de salto* se usan para transferir el control. Este grupo contiene las instrucciones `break`, `continue`, `goto`, `throw`, `return` y `yield`.

La instrucción `try`... `catch` se usa para detectar excepciones que se producen durante la ejecución de un bloque, y la instrucción `try`... `finally` se usa para especificar el código de finalización que siempre se ejecuta, tanto si se ha producido una excepción como si no.

Las instrucciones `checked` y `unchecked` sirven para controlar el contexto de comprobación de desbordamiento para conversiones y operaciones aritméticas de tipo integral.

La instrucción `lock` se usa para obtener el bloqueo de exclusión mutua para un objeto determinado, ejecutar una instrucción y, luego, liberar el bloqueo.

La instrucción `using` se usa para obtener un recurso, ejecutar una instrucción y, luego, eliminar dicho recurso.

A continuación se enumeran los tipos de instrucciones que se pueden usar y se proporciona un ejemplo de cada una.

- Declaración de variable local:

```
static void Declarations(string[] args)
{
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

- Declaración de constante local:

```
static void ConstantDeclarations(string[] args)
{
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

- Instrucción de expresión:

```
static void Expressions(string[] args)
{
    int i;
    i = 123;                // Expression statement
    Console.WriteLine(i);   // Expression statement
    i++;                    // Expression statement
    Console.WriteLine(i);   // Expression statement
}
```

- Instrucción `if`:

```
static void IfStatement(string[] args)
{
    if (args.Length == 0)
    {
        Console.WriteLine("No arguments");
    }
    else
    {
        Console.WriteLine("One or more arguments");
    }
}
```

- Instrucción `switch`:

```
static void SwitchStatement(string[] args)
{
    int n = args.Length;
    switch (n)
    {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine($"{n} arguments");
            break;
    }
}
```

- Instrucción `while`:

```
static void WhileStatement(string[] args)
{
    int i = 0;
    while (i < args.Length)
    {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

- Instrucción `do`:

```
static void DoStatement(string[] args)
{
    string s;
    do
    {
        s = Console.ReadLine();
        Console.WriteLine(s);
    } while (!string.IsNullOrEmpty(s));
}
```

- Instrucción `for`:

```
static void ForStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        Console.WriteLine(args[i]);
    }
}
```

- Instrucción `foreach`:

```
static void ForEachStatement(string[] args)
{
    foreach (string s in args)
    {
        Console.WriteLine(s);
    }
}
```

- Instrucción `break`:

```
static void BreakStatement(string[] args)
{
    while (true)
    {
        string s = Console.ReadLine();
        if (string.IsNullOrEmpty(s))
            break;
        Console.WriteLine(s);
    }
}
```

- Instrucción `continue`:

```
static void ContinueStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        if (args[i].StartsWith("/"))
            continue;
        Console.WriteLine(args[i]);
    }
}
```

- Instrucción `goto`:

```
static void GoToStatement(string[] args)
{
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length)
        goto loop;
}
```

- Instrucción `return`:

```
static int Add(int a, int b)
{
    return a + b;
}
static void ReturnStatement(string[] args)
{
   Console.WriteLine(Add(1, 2));
   return;
}
```

- Instrucción `yield`:

```
static System.Collections.Generic.IEnumerable<int> Range(int start, int end)
{
    for (int i = start; i < end; i++)
    {
        yield return i;
    }
    yield break;
}
static void YieldStatement(string[] args)
{
    foreach (int i in Range(-10,10))
    {
        Console.WriteLine(i);
    }
}
```

- Instrucciones `throw` e instrucciones `try`:

```
static double Divide(double x, double y)
{
    if (y == 0)
        throw new DivideByZeroException();
    return x / y;
}
static void TryCatch(string[] args)
{
    try
    {
        if (args.Length != 2)
        {
            throw new InvalidOperationException("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (InvalidOperationException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        Console.WriteLine("Good bye!");
    }
}
```

- Instrucciones `checked` y `unchecked`:

```
static void CheckedUnchecked(string[] args)
{
    int x = int.MaxValue;
    unchecked
    {
        Console.WriteLine(x + 1);  // Overflow
    }
    checked
    {
        Console.WriteLine(x + 1);  // Exception
    }
}
```

- Instrucción `lock`:

```
class Account
{
    decimal balance;
    private readonly object sync = new object();
    public void Withdraw(decimal amount)
    {
        lock (sync)
        {
            if (amount > balance)
            {
                throw new Exception(
                    "Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

- Instrucción `using`:

```
static void UsingStatement(string[] args)
{
    using (TextWriter w = File.CreateText("test.txt"))
    {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

ANTERIOR SIGUIENTE

Clases y objetos

18/03/2020 • 43 minutes to read • Edit Online

Las *clases* son los tipos más fundamentales de C#. Una clase es una estructura de datos que combina estados (campos) y acciones (métodos y otros miembros de función) en una sola unidad. Una clase proporciona una definición para *instancias* creadas dinámicamente de la clase, también conocidas como *objetos*. Las clases admiten *herencia* y *polimorfismo*, mecanismos por los que las *clases derivadas* pueden extender y especializar *clases base*.

Las clases nuevas se crean mediante declaraciones de clase. Una declaración de clase se inicia con un encabezado que especifica los atributos y modificadores de la clase, el nombre de la clase, la clase base (si se indica) y las interfaces implementadas por la clase. Al encabezado le sigue el cuerpo de la clase, que consta de una lista de declaraciones de miembros escritas entre los delimitadores `{` y `}`.

En el código siguiente se muestra una declaración de una clase simple denominada `Point`:

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Las instancias de clases se crean mediante el operador `new`, que asigna memoria para una nueva instancia, invoca un constructor para inicializar la instancia y devuelve una referencia a la instancia. Las instrucciones siguientes crean dos objetos Point y almacenan las referencias en esos objetos en dos variables:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

La memoria ocupada por un objeto se reclama automáticamente cuando el objeto ya no es accesible. En C#, no es necesario ni posible desasignar objetos de forma explícita.

Miembros

Los miembros de una clase son miembros estáticos o miembros de instancia. Los miembros estáticos pertenecen a clases y los miembros de instancia pertenecen a objetos (instancias de clases).

En la lista siguiente se proporciona una visión general de los tipos de miembros que puede contener una clase.

- Constantes
  - Valores constantes asociados a la clase
- Campos
  - Variables de la clase
- Métodos
  - Cálculos y acciones que pueden realizarse mediante la clase
- Propiedades
  - Acciones asociadas a la lectura y escritura de propiedades con nombre de la clase

- Indizadores
  - Acciones asociadas a la indexación de instancias de la clase como una matriz
- Eventos
  - Notificaciones que puede generar la clase
- Operadores
  - Conversiones y operadores de expresión admitidos por la clase
- Constructores
  - Acciones necesarias para inicializar instancias de la clase o la clase propiamente dicha
- Finalizadores
  - Acciones que deben realizarse antes de que las instancias de la clase se descarten de forma permanente
- Tipos
  - Tipos anidados declarados por la clase

Accesibilidad

Cada miembro de una clase tiene asociada una accesibilidad, que controla las regiones del texto del programa que pueden acceder al miembro. Existen seis formas de accesibilidad posibles. A continuación se resumen los modificadores de acceso.

- `public`
  - El acceso no está limitado.
- `protected`
  - El acceso está limitado a esta clase o a las clases derivadas de esta clase.
- `internal`
  - El acceso está limitado al ensamblado actual (.exe, .dll, etc.).
- `protected internal`
  - El acceso está limitado a la clase contenedora, las clases derivadas de la clase contenedora, o bien las clases dentro del mismo ensamblado.
- `private`
  - El acceso está limitado a esta clase.
- `private protected`
  - El acceso está limitado a la clase contenedora o las clases derivadas del tipo contenedor dentro del mismo ensamblado.

Parámetros de tipo

Una definición de clase puede especificar un conjunto de parámetros de tipo poniendo tras el nombre de clase una lista de nombres de parámetro de tipo entre corchetes angulares. Los parámetros de tipo pueden usarse luego en el cuerpo de las declaraciones de clase para definir a los miembros de la clase. En el ejemplo siguiente, los parámetros de tipo de `Pair` son `TFirst` y `TSecond`:

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

Un tipo de clase que se declara para tomar parámetros de tipo se conoce como *tipo de clase genérica*. Los tipos de estructura, interfaz y delegado también pueden ser genéricos. Cuando se usa la clase genérica, se deben proporcionar argumentos de tipo para cada uno de los parámetros de tipo:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

Un tipo genérico con argumentos de tipo proporcionado, como `Pair<int,string>` anteriormente, se conoce como *tipo construido*.

Clases base

Una declaración de clase puede especificar una clase base colocando después del nombre de clase y los parámetros de tipo dos puntos seguidos del nombre de la clase base. Omitir una especificación de la clase base es igual que derivarla del tipo `object`. En el ejemplo siguiente, la clase base de `Point3D` es `Point` y la clase base de `Point` es `object`:

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
public class Point3D: Point
{
    public int z;
    public Point3D(int x, int y, int z) :
        base(x, y)
    {
        this.z = z;
    }
}
```

Una clase hereda a los miembros de su clase base. La herencia significa que una clase contiene implícitamente todos los miembros de su clase base, excepto la instancia y los constructores estáticos, y los finalizadores de la clase base. Una clase derivada puede agregar nuevos miembros a aquellos de los que hereda, pero no puede quitar la definición de un miembro heredado. En el ejemplo anterior, `Point3D` hereda los campos `x` y `y` de `Point` y cada instancia de `Point3D` contiene tres campos: `x`, `y` y `z`.

Existe una conversión implícita de un tipo de clase a cualquiera de sus tipos de clase base. Una variable de un tipo de clase puede hacer referencia a una instancia de esa clase o a una instancia de cualquier clase derivada. Por ejemplo, dadas las declaraciones de clase anteriores, una variable de tipo `Point` puede hacer referencia a una instancia de `Point` o `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Campos

Un *campo* es una variable que está asociada con una clase o a una instancia de una clase.

Un campo declarado con el modificador "static" define un campo estático. Un campo estático identifica exactamente una ubicación de almacenamiento. Independientemente del número de instancias de una clase que se creen, solo hay una única copia de un campo estático.

Un campo declarado sin el modificador "static" define un campo de instancia. Cada instancia de una clase contiene

una copia independiente de todos los campos de instancia de esa clase.

En el ejemplo siguiente, cada instancia de la clase `Color` tiene una copia independiente de los campos de instancia `r`, `g` y `b`, pero solo hay una copia de los campos estáticos `Black`, `White`, `Red`, `Green` y `Blue`:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;
    public Color(byte r, byte g, byte b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

Como se muestra en el ejemplo anterior, los *campos de solo lectura* se puede declarar con un modificador `readonly`. La asignación a un campo `readonly` solo se puede producir como parte de la declaración del campo o en un constructor de la misma clase.

Métodos

Un *método* es un miembro que implementa un cálculo o una acción que puede realizar un objeto o una clase. A los *métodos estáticos* se accede a través de la clase. A los *métodos de instancia* se accede a través de instancias de la clase.

Los métodos pueden tener una lista de *parámetros*, que representan valores o referencias a variables que se pasan al método, y un *tipo de valor devuelto*, que especifica el tipo del valor calculado y devuelto por el método. El tipo de valor devuelto de un método es `void` si no devuelve un valor.

Al igual que los tipos, los métodos también pueden tener un conjunto de parámetros de tipo, para lo cuales se deben especificar argumentos de tipo cuando se llama al método. A diferencia de los tipos, los argumentos de tipo a menudo se pueden deducir de los argumentos de una llamada al método y no es necesario proporcionarlos explícitamente.

La *signatura* de un método debe ser única en la clase en la que se declara el método. La signatura de un método se compone del nombre del método, el número de parámetros de tipo y el número, los modificadores y los tipos de sus parámetros. La signatura de un método no incluye el tipo de valor devuelto.

Parámetros

Los parámetros se usan para pasar valores o referencias a variables a métodos. Los parámetros de un método obtienen sus valores reales de los *argumentos* que se especifican cuando se invoca el método. Hay cuatro tipos de parámetros: parámetros de valor, parámetros de referencia, parámetros de salida y matrices de parámetros.

Un *parámetro de valor* se usa para pasar argumentos de entrada. Un parámetro de valor corresponde a una variable local que obtiene su valor inicial del argumento que se ha pasado para el parámetro. Las modificaciones de un parámetro de valor no afectan el argumento que se ha pasado para el parámetro.

Los parámetros de valor pueden ser opcionales; se especifica un valor predeterminado para que se puedan omitir los argumentos correspondientes.

Un *parámetro de referencia* se usa para pasar argumentos mediante una referencia. El argumento pasado para un parámetro de referencia debe ser una variable con un valor definitivo, y durante la ejecución del método, el

parámetro de referencia representa la misma ubicación de almacenamiento que la variable del argumento. Un parámetro de referencia se declara con el modificador `ref`. En el ejemplo siguiente se muestra el uso de parámetros `ref`.

```
using System;
class RefExample
{
    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void SwapExample()
    {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine($"{i} {j}");    // Outputs "2 1"
    }
}
```

Un *parámetro de salida* se usa para pasar argumentos mediante una referencia. Es similar a un parámetro de referencia, excepto que no necesita que asigne un valor explícitamente al argumento proporcionado por el autor de la llamada. Un parámetro de salida se declara con el modificador `out`. En el siguiente ejemplo se muestra el uso de los parámetros `out` con la sintaxis que se ha presentado en C# 7.

```
    using System;
    class OutExample
    {
        static void Divide(int x, int y, out int result, out int remainder)
        {
            result = x / y;
            remainder = x % y;
        }
        public static void OutUsage()
        {
            Divide(10, 3, out int res, out int rem);
            Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"
        }
    }
}
```

Una *matriz de parámetros* permite que se pasen a un método un número variable de argumentos. Una matriz de parámetros se declara con el modificador `params`. Solo el último parámetro de un método puede ser una matriz de parámetros y el tipo de una matriz de parámetros debe ser un tipo de matriz unidimensional. Los métodos Write y WriteLine de la clase System.Console son buenos ejemplos de uso de la matriz de parámetros. Se declaran de la manera siguiente.

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

Dentro de un método que usa una matriz de parámetros, la matriz de parámetros se comporta exactamente igual que un parámetro normal de un tipo de matriz. Pero en una invocación de un método con una matriz de parámetros, es posible pasar un único argumento del tipo de matriz de parámetros o cualquier número de argumentos del tipo de elemento de la matriz de parámetros. En este caso, una instancia de matriz se e inicializa

automáticamente con los argumentos dados. Este ejemplo

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

es equivalente a escribir lo siguiente.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Cuerpo del método y variables locales

El cuerpo de un método especifica las instrucciones que se ejecutarán cuando se invoca el método.

Un cuerpo del método puede declarar variables que son específicas de la invocación del método. Estas variables se denominan *variables locales*. Una declaración de variable local especifica un nombre de tipo, un nombre de variable y, posiblemente, un valor inicial. En el ejemplo siguiente se declara una variable local `i` con un valor inicial de cero y una variable local `j` sin ningún valor inicial.

```
using System;
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i = i + 1;
        }
    }
}
```

C# requiere que se *asigne definitivamente* una variable local antes de que se pueda obtener su valor. Por ejemplo, si la declaración de `i` anterior no incluyera un valor inicial, el compilador notificaría un error con los usos posteriores de `i` porque `i` no se asignaría definitivamente en esos puntos del programa.

Puede usar una instrucción `return` para devolver el control a su llamador. En un método que devuelve `void`, las instrucciones `return` no pueden especificar una expresión. En un método que devuelve valores distintos de void, las instrucciones `return` deben incluir una expresión que calcula el valor devuelto.

Métodos estáticos y de instancia

Un método declarado con un modificador static es un *método estático*. Un método estático no opera en una instancia específica y solo puede acceder directamente a miembros estáticos.

Un método declarado sin un modificador static es un *método de instancia*. Un método de instancia opera en una instancia específica y puede acceder a miembros estáticos y de instancia. Se puede acceder explícitamente a la instancia en la que se invoca un método de instancia como `this`. Es un error hacer referencia a `this` en un método estático.

La siguiente clase `Entity` tiene miembros estáticos y de instancia.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity()
    {
        serialNo = nextSerialNo++;
    }
    public int GetSerialNo()
    {
        return serialNo;
    }
    public static int GetNextSerialNo()
    {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value)
    {
        nextSerialNo = value;
    }
}
```

Cada instancia de `Entity` contiene un número de serie (y, probablemente, otra información que no se muestra aquí). El constructor `Entity` (que es como un método de instancia) inicializa la nueva instancia con el siguiente número de serie disponible. Como el constructor es un miembro de instancia, se le permite acceder al campo de instancia `serialNo` y al campo estático `nextSerialNo`.

Los métodos estáticos `GetNextSerialNo` y `SetNextSerialNo` pueden acceder al campo estático `nextSerialNo`, pero sería un error para ellas acceder directamente al campo de instancia `serialNo`.

En el ejemplo siguiente se muestra el uso de la clase Entity.

```
using System;
class EntityExample
{
    public static void Usage()
    {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());            // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());            // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());    // Outputs "1002"
    }
}
```

Los métodos estáticos `SetNextSerialNo` y `GetNextSerialNo` se invocan en la clase, mientras que el método de instancia `GetSerialNo` se invoca en instancias de la clase.

Métodos virtual, de reemplazo y abstracto

Cuando una declaración de método de instancia incluye un modificador `virtual`, se dice que el método es un *método virtual*. Cuando no existe un modificador virtual, se dice que el método es un *método no virtual*.

Cuando se invoca un método virtual, el *tipo en tiempo de ejecución* de la instancia para la que tiene lugar esa invocación determina la implementación del método real que se invocará. En una invocación de método no virtual, el *tipo en tiempo de compilación* de la instancia es el factor determinante.

Un método virtual puede ser *reemplazado* en una clase derivada. Cuando una declaración de método de instancia incluye un modificador "override", el método reemplaza un método virtual heredado con la misma signatura. Mientras que una declaración de método virtual introduce un método nuevo, una declaración de método de

reemplazo especializa un método virtual heredado existente proporcionando una nueva implementación de ese método.

Un *método abstracto* es un método virtual sin implementación. Un método abstracto se declara con el modificador "abstract" y solo se permite en una clase que también se declare abstracta. Un método abstracto debe reemplazarse en todas las clases derivadas no abstractas.

En el ejemplo siguiente se declara una clase abstracta, `Expression`, que representa un nodo de árbol de expresión y tres clases derivadas, `Constant`, `VariableReference` y `Operation`, que implementan nodos de árbol de expresión para constantes, referencias a variables y operaciones aritméticas. (Este ejemplo es similar a los tipos de árbol de expresión, pero no debe confundirse con ellos).

```
using System;
using System.Collections.Generic;
public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string,object> vars);
}
public class Constant: Expression
{
    double value;
    public Constant(double value)
    {
        this.value = value;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        return value;
    }
}
public class VariableReference: Expression
{
    string name;
    public VariableReference(string name)
    {
        this.name = name;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        object value = vars[name];
        if (value == null)
        {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}
public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;
    public Operation(Expression left, char op, Expression right)
    {
        this.left = left;
        this.op = op;
        this.right = right;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```

Las cuatro clases anteriores se pueden usar para modelar expresiones aritméticas. Por ejemplo, usando instancias de estas clases, la expresión `x + 3` se puede representar de la manera siguiente.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

El método `Evaluate` de una instancia `Expression` se invoca para evaluar la expresión determinada y generar un valor `double`. El método toma un argumento `Dictionary` que contiene nombres de variables (como claves de las entradas) y valores (como valores de las entradas). Como `Evaluate` es un método abstracto, las clases no abstractas que derivan de `Expression` deben invalidar `Evaluate`.

Una implementación de `Constant` de `Evaluate` simplemente devuelve la constante almacenada. Una implementación de `VariableReference` busca el nombre de variable en el diccionario y devuelve el valor resultante. Una implementación de `Operation` evalúa primero los operandos izquierdo y derecho (mediante la invocación recursiva de sus métodos `Evaluate`) y luego realiza la operación aritmética correspondiente.

El siguiente programa usa las clases `Expression` para evaluar la expresión `x * (y + 2)` para los distintos valores de `x` y `y`.

```
using System;
using System.Collections.Generic;
class InheritanceExample
{
    public static void ExampleUsage()
    {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Dictionary<string,object> vars = new Dictionary<string, object>();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));  // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));  // Outputs "16.5"
    }
}
```

Sobrecarga de métodos

La *sobrecarga* de métodos permite que varios métodos de la misma clase tengan el mismo nombre mientras tengan signaturas únicas. Al compilar una invocación de un método sobrecargado, el compilador usa la *resolución de sobrecarga* para determinar el método concreto que de invocará. La resolución de sobrecarga busca el método que mejor coincida con los argumentos o informa de un error si no se puede encontrar ninguna mejor coincidencia. En el ejemplo siguiente se muestra la resolución de sobrecarga en vigor. El comentario para cada invocación del método `UsageExample` muestra qué método se invoca.

```
using System;
class OverloadingExample
{
    static void F()
    {
        Console.WriteLine("F()");
    }
    static void F(object x)
    {
        Console.WriteLine("F(object)");
    }
    static void F(int x)
    {
        Console.WriteLine("F(int)");
    }
    static void F(double x)
    {
        Console.WriteLine("F(double)");
    }
    static void F<T>(T x)
    {
        Console.WriteLine("F<T>(T)");
    }
    static void F(double x, double y)
    {
        Console.WriteLine("F(double, double)");
    }
    public static void UsageExample()
    {
        F();            // Invokes F()
        F(1);           // Invokes F(int)
        F(1.0);         // Invokes F(double)
        F("abc");       // Invokes F<string>(string)
        F((double)1);   // Invokes F(double)
        F((object)1);   // Invokes F(object)
        F<int>(1);      // Invokes F<int>(int)
        F(1, 1);        // Invokes F(double, double)
    }
}
```

Tal como se muestra en el ejemplo, un método determinado siempre se puede seleccionar mediante la conversión explícita de los argumentos en los tipos de parámetros exactos o el suministro explícito de los argumentos de tipo.

Otros miembros de función

Los miembros que contienen código ejecutable se conocen colectivamente como *miembros de función* de una clase. En la sección anterior se describen los métodos, que son los tipos principales de los miembros de función. En esta sección se describen los otros tipos de miembros de función admitidos por C#: constructores, propiedades, indexadores, eventos, operadores y finalizadores.

En el ejemplo siguiente se muestra una clase genérica llamada `MyList<T>`, que implementa una lista creciente de objetos. La clase contiene varios ejemplos de los tipos más comunes de miembros de función.

> **NOTE**
>
> Este ejemplo crea una clase `MyList`, que no es la misma que la clase System.Collections.Generic.List<T> estándar de .NET. Ilustra los conceptos necesarios para esta guía, pero no sustituye a esa clase.

```
public class MyList<T>
{
    // Constant
```

```
    const int defaultCapacity = 4;

    // Fields
    T[] items;
    int count;

    // Constructor
    public MyList(int capacity = defaultCapacity)
    {
        items = new T[capacity];
    }

    // Properties
    public int Count => count;

    public int Capacity
    {
        get { return items.Length; }
        set
        {
            if (value < count) value = count;
            if (value != items.Length)
            {
                T[] newItems = new T[value];
                Array.Copy(items, 0, newItems, 0, count);
                items = newItems;
            }
        }
    }

    // Indexer
    public T this[int index]
    {
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods
    public void Add(T item)
    {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() =>
        Changed?.Invoke(this, EventArgs.Empty);

    public override bool Equals(object other) =>
        Equals(this, other as MyList<T>);

    static bool Equals(MyList<T> a, MyList<T> b)
    {
        if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
        if (Object.ReferenceEquals(b, null) || a.count != b.count)
            return false;
        for (int i = 0; i < a.count; i++)
        {
            if (!object.Equals(a.items[i], b.items[i]))
            {
                return false;
            }
```

```
        }
    return true;
    }

    // Event
    public event EventHandler Changed;

    // Operators
    public static bool operator ==(MyList<T> a, MyList<T> b) =>
        Equals(a, b);

    public static bool operator !=(MyList<T> a, MyList<T> b) =>
        !Equals(a, b);
}
```

Constructores

C# admite constructores de instancia y estáticos. Un *constructor de instancia* es un miembro que implementa las acciones necesarias para inicializar una instancia de una clase. Un *constructor estático* es un miembro que implementa las acciones necesarias para inicializar una clase en sí misma cuando se carga por primera vez.

Un constructor se declara como un método sin ningún tipo de valor devuelto y el mismo nombre que la clase contenedora. Si una declaración de constructor incluye un modificador "static", declara un constructor estático. De lo contrario, declara un constructor de instancia.

Los constructores de instancia pueden sobrecargarse y tener parámetros opcionales. Por ejemplo, la clase `MyList<T>` declara un constructor de instancia con único parámetro `int` opcional. Los constructores de instancia se invocan mediante el operador `new`. Las siguientes instrucciones asignan dos instancias `MyList<string>` mediante el constructor de la clase `MyList` con y sin el argumento opcional.

```
MyList<string> list1 = new MyList<string>();
MyList<string> list2 = new MyList<string>(10);
```

A diferencia de otros miembros, los constructores de instancia no se heredan y una clase no tiene ningún constructor de instancia que no sea el que se declara realmente en la clase. Si no se proporciona ningún constructor de instancia para una clase, se proporciona automáticamente uno vacío sin ningún parámetro.

Propiedades

Las *propiedades* son una extensión natural de los campos. Ambos son miembros con nombre con tipos asociados y la sintaxis para acceder a los campos y las propiedades es la misma. Pero a diferencia de los campos, las propiedades no denotan ubicaciones de almacenamiento. Las propiedades tienen *descriptores de acceso* que especifican las instrucciones que se ejecutan cuando se leen o escriben sus valores.

Una propiedad se declara como un campo, salvo que la declaración finaliza con un descriptor de acceso get y un descriptor de acceso set escrito entre los delimitadores `{` y `}` en lugar de finalizar en un punto y coma. Una propiedad que tiene un descriptor de acceso get y un descriptor de acceso set es una *propiedad de lectura y escritura*, una propiedad que tiene solo un descriptor de acceso get es una *propiedad de solo lectura* y una propiedad que tiene solo un descriptor de acceso set es una *propiedad de solo escritura*.

Un descriptor de acceso get corresponde a un método sin parámetros con un valor devuelto del tipo de propiedad. Excepto como destino de una asignación, cuando se hace referencia a una propiedad en una expresión, el descriptor de acceso get de la propiedad se invoca para calcular el valor de la propiedad.

Un descriptor de acceso set corresponde a un método con un solo parámetro denominado value y ningún tipo de valor devuelto. Cuando se hace referencia a una propiedad como el destino de una asignación o como el operando de ++ o--, el descriptor de acceso set se invoca con un argumento que proporciona el nuevo valor.

La clase `MyList<T>` declara dos propiedades, `Count` y `Capacity`, que son de solo lectura y de lectura y escritura,

respectivamente. El código siguiente es un ejemplo de uso de estas propiedades:

```
MyList<string> names = new MyList<string>();
names.Capacity = 100;   // Invokes set accessor
int i = names.Count;    // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

De forma similar a los campos y métodos, C# admite propiedades de instancia y propiedades estáticas. Las propiedades estáticas se declaran con el modificador "static", y las propiedades de instancia se declaran sin él.

Los descriptores de acceso de una propiedad pueden ser virtuales. Cuando una declaración de propiedad incluye un modificador `virtual`, `abstract` o `override`, se aplica a los descriptores de acceso de la propiedad.

Indizadores

Un *indexador* es un miembro que permite indexar de la misma manera que una matriz. Un indexador se declara como una propiedad, excepto por el hecho que el nombre del miembro es `this`, seguido por una lista de parámetros que se escriben entre los delimitadores `[` y `]`. Los parámetros están disponibles en los descriptores de acceso del indexador. De forma similar a las propiedades, los indexadores pueden ser lectura y escritura, de solo lectura y de solo escritura, y los descriptores de acceso de un indexador pueden ser virtuales.

La clase `MyList<T>` declara un único indexador de lectura y escritura que toma un parámetro `int`. El indexador permite indexar instancias de `MyList<T>` con valores `int`. Por ejemplo:

```
MyList<string> names = new MyList<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Los indexadores se pueden sobrecargar, lo que significa que una clase puede declarar varios indexadores siempre y cuando el número o los tipos de sus parámetros sean diferentes.

Events

Un *evento* es un miembro que permite que una clase u objeto proporcionen notificaciones. Un evento se declara como un campo, excepto por el hecho de que la declaración incluye una palabra clave event y el tipo debe ser un tipo delegado.

Dentro de una clase que declara un miembro de evento, el evento se comporta como un campo de un tipo delegado (siempre que el evento no sea abstracto y no declare descriptores de acceso). El campo almacena una referencia a un delegado que representa los controladores de eventos que se han agregado al evento. Si no existen controladores de eventos, el campo es `null`.

La clase `MyList<T>` declara un único miembro de evento llamado `Changed`, lo que indica que se ha agregado un nuevo elemento a la lista. El método virtual `OnChanged` genera el evento cambiado, y comprueba primero si el evento es `null` (lo que significa que no existen controladores). La noción de generar un evento es equivalente exactamente a invocar el delegado representado por el evento; por lo tanto, no hay ninguna construcción especial de lenguaje para generar eventos.

Los clientes reaccionan a los eventos mediante *controladores de eventos*. Los controladores de eventos se asocian mediante el operador `+=` y se quitan con el operador `-=`. En el ejemplo siguiente se asocia un controlador de eventos con el evento `Changed` de un objeto `MyList<string>`.

```
class EventExample
{
    static int changeCount;
    static void ListChanged(object sender, EventArgs e)
    {
        changeCount++;
    }
    public static void Usage()
    {
        MyList<string> names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);  // Outputs "3"
    }
}
```

Para escenarios avanzados donde se quiere controlar el almacenamiento subyacente de un evento, una declaración de evento puede proporcionar de forma explícita los descriptores de acceso `add` y `remove`, que son similares al descriptor de acceso `set` de una propiedad.

Operadores

Un *operador* es un miembro que define el significado de aplicar un operador de expresión determinado a las instancias de una clase. Se pueden definir tres tipos de operadores: operadores unarios, operadores binarios y operadores de conversión. Todos los operadores se deben declarar como `public` y `static`.

La clase `MyList<T>` declara dos operadores, `operator ==` y `operator !=`, y de este modo proporciona un nuevo significado a expresiones que aplican esos operadores a instancias `MyList`. En concreto, los operadores definen la igualdad de dos instancias `MyList<T>` como la comparación de cada uno de los objetos contenidos con sus métodos Equals. En el ejemplo siguiente se usa el operador `==` para comparar dos instancias `MyList<int>`.

```
MyList<int> a = new MyList<int>();
a.Add(1);
a.Add(2);
MyList<int> b = new MyList<int>();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b);  // Outputs "True"
b.Add(3);
Console.WriteLine(a == b);  // Outputs "False"
```

El primer objeto `Console.WriteLine` genera `True` porque las dos listas contienen el mismo número de objetos con los mismos valores en el mismo orden. Si `MyList<T>` no hubiera definido `operator ==`, el primer objeto `Console.WriteLine` habría generado `False` porque `a` y `b` hacen referencia a diferentes instancias de `MyList<int>`.

Finalizadores

Un *finalizador* es un miembro que implementa las acciones necesarias para finalizar una instancia de una clase. Los finalizadores no pueden tener parámetros, no pueden tener modificadores de accesibilidad y no se pueden invocar de forma explícita. El finalizador de una instancia se invoca automáticamente durante la recolección de elementos no utilizados.

El recolector de elementos no utilizados tiene una amplia libertad para decidir cuándo debe recolectar objetos y ejecutar finalizadores. En concreto, los intervalos de las invocaciones de finalizador no son deterministas y los finalizadores se pueden ejecutar en cualquier subproceso. Por estas y otras razones, las clases deben implementar finalizadores solo cuando no haya otras soluciones que sean factibles.

La instrucción `using` proporciona un mejor enfoque para la destrucción de objetos.

ANTERIOR SIGUIENTE

Matrices

18/03/2020 • 5 minutes to read • Edit Online

Una ***matriz*** es una estructura de datos que contiene un número de variables a las que se accede mediante índices calculados. Las variables contenidas en una matriz, denominadas también ***elementos*** de la matriz, son todas del mismo tipo y este tipo se conoce como ***tipo de elemento*** de la matriz.

Los tipos de matriz son tipos de referencia, y la declaración de una variable de matriz simplemente establece un espacio reservado para una referencia a una instancia de matriz. Las instancias de matriz reales se crean dinámicamente en tiempo de ejecución mediante el operador `new`. La nueva operación especifica la ***longitud*** de la nueva instancia de matriz, que luego se fija para la vigencia de la instancia. Los índices de los elementos de una matriz van de `0` a `Length - 1`. El operador `new` inicializa automáticamente los elementos de una matriz a su valor predeterminado, que, por ejemplo, es cero para todos los tipos numéricos y `null` para todos los tipos de referencias.

En el ejemplo siguiente se crea una matriz de elementos `int`, se inicializa la matriz y se imprime el contenido de la matriz.

```
using System;
class ArrayExample
{
    static void Main()
    {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++)
        {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++)
        {
            Console.WriteLine($"a[{i}] = {a[i]}");
        }
    }
}
```

Este ejemplo se crea y se pone en funcionamiento en una ***matriz unidimensional***. C# también admite ***matrices multidimensionales***. El número de dimensiones de un tipo de matriz, conocido también como ***rango*** del tipo de matriz, es una más el número de comas escritas entre los corchetes del tipo de matriz. En el ejemplo siguiente se asignan una matriz unidimensional, bidimensional y tridimensional, respectivamente.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

La matriz `a1` contiene 10 elementos, la matriz `a2` 50 (10 × 5) elementos y la matriz `a3` 100 (10 × 5 × 2) elementos. El tipo de elemento de una matriz puede ser cualquiera, incluido un tipo de matriz. Una matriz con elementos de un tipo de matriz a veces se conoce como ***matriz escalonada*** porque las longitudes de las matrices de elementos no tienen que ser iguales. En el ejemplo siguiente se asigna una matriz de matrices de `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

La primera línea crea una matriz con tres elementos, cada uno de tipo `int[]` y cada uno con un valor inicial de `null`. Las líneas posteriores inicializan entonces los tres elementos con referencias a instancias de matriz individuales de longitud variable.

El nuevo operador permite especificar los valores iniciales de los elementos de matriz mediante un ***inicializador de matriz***, que es una lista de las expresiones escritas entre los delimitadores `{` y `}`. En el ejemplo siguiente se asigna e inicializa un tipo `int[]` con tres elementos.

```
int[] a = new int[] {1, 2, 3};
```

La longitud de la matriz se deduce del número de expresiones entre { y }. Las declaraciones de variable local y campo se pueden acortar más para que así no sea necesario reformular el tipo de matriz.

```
int[] a = {1, 2, 3};
```

Los dos ejemplos anteriores son equivalentes al código siguiente:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

ANTERIOR SIGUIENTE

Interfaces

18/03/2020 • 3 minutes to read • Edit Online

Una *interfaz* define un contrato que se puede implementar mediante clases y structs. Una interfaz puede contener métodos, propiedades, eventos e indexadores. Una interfaz no proporciona implementaciones de los miembros que define, simplemente especifica los miembros que se deben proporcionar mediante clases o estructuras que implementan la interfaz.

Las interfaces pueden usar ***herencia múltiple***. En el ejemplo siguiente, la interfaz `IComboBox` hereda de `ITextBox` y `IListBox`.

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

Las clases y los structs pueden implementar varias interfaces. En el ejemplo siguiente, la clase `EditBox` implementa `IControl` y `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}
public class EditBox: IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

Cuando una clase o un struct implementan una interfaz determinada, las instancias de esa clase o struct se pueden convertir implícitamente a ese tipo de interfaz. Por ejemplo

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

En casos donde una instancia no se conoce estáticamente para implementar una interfaz determinada, se pueden usar conversiones de tipo dinámico. Por ejemplo, las siguientes instrucciones usan conversiones de tipo dinámico para obtener las implementaciones de `IControl` y `IDataBound` de un objeto. Dado que el tipo real en tiempo de ejecución del objeto es `EditBox`, las conversiones se realizan correctamente.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

En la clase `EditBox` anterior, el método `Paint` de la interfaz `IControl` y el método `Bind` de la interfaz `IDataBound` se implementan mediante miembros públicos. C# también admite ***implementaciones de miembros de interfaz*** explícitos, lo que permite que la clase o el struct eviten que los miembros se conviertan en públicos. Una implementación de miembro de interfaz explícito se escribe con el nombre de miembro de interfaz completo. Por ejemplo, la clase `EditBox` podría implementar los métodos `IControl.Paint` y `IDataBound.Bind` mediante implementaciones de miembros de interfaz explícitos del modo siguiente.

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() { }
    void IDataBound.Bind(Binder b) { }
}
```

Solo se puede acceder a los miembros de interfaz explícitos mediante el tipo de interfaz. Por ejemplo, la implementación de `IControl.Paint` proporcionada por la clase EditBox anterior solo se puede invocar si se convierte primero la referencia `EditBox` al tipo de interfaz `IControl`.

```
EditBox editBox = new EditBox();
editBox.Paint();            // Error, no such method
IControl control = editBox;
control.Paint();            // Ok
```

ANTERIOR SIGUIENTE

Delegados

18/03/2020 • 3 minutes to read • Edit Online

Un ***tipo de delegado*** representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son similares al concepto de punteros de función de otros lenguajes. A diferencia de los punteros de función, los delegados están orientados a objetos y tienen seguridad de tipos.

En el siguiente ejemplo se declara y usa un tipo de delegado denominado `Function`.

```
using System;
delegate double Function(double x);
class Multiplier
{
    double factor;
    public Multiplier(double factor)
    {
        this.factor = factor;
    }
    public double Multiply(double x)
    {
        return x * factor;
    }
}
class DelegateExample
{
    static double Square(double x)
    {
        return x * x;
    }
    static double[] Apply(double[] a, Function f)
    {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }
    static void Main()
    {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

Una instancia del tipo de delegado `Function` puede hacer referencia a cualquier método que tome un argumento `double` y devuelva un valor `double`. El método `Apply` aplica una función dada a los elementos de un argumento `double[]`, y devuelve un valor `double[]` con los resultados. En el método `Main`, `Apply` se usa para aplicar tres funciones diferentes a un valor `double[]`.

Un delegado puede hacer referencia a un método estático (como `Square` o `Math.Sin` en el ejemplo anterior) o un método de instancia (como `m.Multiply` en el ejemplo anterior). Un delegado que hace referencia a un método de instancia también hace referencia a un objeto determinado y, cuando se invoca el método de instancia a través del delegado, ese objeto se convierte en `this` en la invocación.

Los delegados también se pueden crear mediante funciones anónimas, que son "métodos insertados" que se crean al declararlos. Las funciones anónimas pueden ver las variables locales de los métodos adyacentes. En el ejemplo siguiente no se crea una clase:

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

Un delegado no sabe ni necesita conocer la clase del método al que hace referencia; lo único que importa es que el método al que se hace referencia tenga los mismos parámetros y el tipo de valor devuelto que el delegado.

ANTERIOR SIGUIENTE

Atributos

18/03/2020 • 3 minutes to read • Edit Online

Los tipos, los miembros y otras entidades en un programa de C # admiten modificadores que controlan ciertos aspectos de su comportamiento. Por ejemplo, la accesibilidad de un método se controla mediante los modificadores `public`, `protected`, `internal` y `private`. C # generaliza esta funcionalidad de manera que los tipos de información declarativa definidos por el usuario se puedan adjuntar a las entidades del programa y recuperarse en tiempo de ejecución. Los programas especifican esta información declarativa adicional mediante la definición y el uso de ***atributos***.

En el ejemplo siguiente se declara un atributo `HelpAttribute` que se puede colocar en entidades de programa para proporcionar vínculos a la documentación asociada.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;
    public HelpAttribute(string url)
    {
        this.url = url;
    }

    public string Url => url;

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

Todas las clases de atributos se derivan de la clase base Attribute proporcionada por la biblioteca estándar. Los atributos se pueden aplicar proporcionando su nombre, junto con cualquier argumento, entre corchetes, justo antes de la declaración asociada. Si el nombre de un atributo termina en `Attribute`, esa parte del nombre se puede omitir cuando se hace referencia al atributo. Por ejemplo, `HelpAttribute` se puede usar de la manera siguiente.

```
[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/attributes")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/attributes",
    Topic = "Display")]
    public void Display(string text) {}
}
```

En este ejemplo se adjunta un atributo `HelpAttribute` a la clase `Widget`. También se agrega otro atributo `HelpAttribute` al método `Display` en la clase. Los constructores públicos de una clase de atributos controlan la información que se debe proporcionar cuando el atributo se adjunta a una entidad de programa. Se puede proporcionar información adicional haciendo referencia a las propiedades públicas de lectura y escritura de la clase de atributos (como la referencia a la propiedad `Topic` usada anteriormente).

Los metadatos definidos por atributos pueden leerse y manipularse en tiempo de ejecución mediante reflexión. Cuando se solicita un atributo determinado mediante esta técnica, se invoca al constructor de la clase de atributos

con la información proporcionada en el origen del programa y se devuelve la instancia de atributo resultante. Si se proporciona información adicional mediante propiedades, dichas propiedades se establecen en los valores dados antes de devolver la instancia del atributo.

El siguiente ejemplo de código demuestra cómo obtener las `HelpAttribute` instancias asociadas a la clase `Widget` y su método `Display`.

```
Type widgetType = typeof(Widget);

//Gets every HelpAttribute defined for the Widget type
object[] widgetClassAttributes = widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic : {attr.Topic}");
}

System.Reflection.MethodInfo displayMethod = widgetType.GetMethod(nameof(Widget.Display));

//Gets every HelpAttribute defined for the Widget.Display method
object[] displayMethodAttributes = displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic : {attr.Topic}");
}

Console.ReadLine();
```

ANTERIOR

Novedades de C# 8.0

12/05/2020 • 30 minutes to read • Edit Online

C# 8.0 agrega las siguientes características y mejoras al lenguaje C#:

- Miembros de solo lectura
- Métodos de interfaz predeterminados
- Mejoras de coincidencia de patrones:
  - Expresiones switch
  - Patrones de propiedades
  - Patrones de tupla
  - Patrones posicionales
- Declaraciones using
- Funciones locales estáticas
- Estructuras ref descartables
- Tipos de referencia que aceptan valores null
- Secuencias asincrónicas
- Asincrónica descartable
- Índices y rangos
- Asignación de uso combinado de NULL
- Tipos construidos no administrados
- Stackalloc en expresiones anidadas
- Mejora de las cadenas textuales interpoladas

C# 8.0 se admite en **.NET Core 3.x** y **.NET Standard 2.1**. Para obtener más información, vea Control de versiones del lenguaje C#.

En el resto de este artículo se describen brevemente estas características. Cuando hay disponibles artículos detallados, se proporcionan vínculos a esos tutoriales e introducciones. Puede explorar estas características en su entorno mediante la herramienta global `dotnet try`:

1. Instale la herramienta global dotnet-try.
2. Clone el repositorio dotnet/try-samples.
3. Establezca el directorio actual en el subdirectorio *csharp8* para el repositorio *try-samples*.
4. Ejecute `dotnet try`.

Miembros de solo lectura

Puede aplicar el modificador `readonly` a los miembros de una estructura. Indica que el miembro no modifica el estado. Resulta más pormenorizado que aplicar el modificador `readonly` a una declaración `struct`. Tenga en cuenta el siguiente struct mutable:

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"({X}, {Y}) is {Distance} from the origin";
}
```

Al igual que la mayoría de las estructuras, el método `ToString()` no modifica el estado. Para indicar eso, podría agregar el modificador `readonly` a la declaración de `ToString()`:

```
public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
```

El cambio anterior genera una advertencia del compilador, porque `ToString` accede a la propiedad `Distance`, que no está marcada como `readonly`:

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an
implicit copy of 'this'
```

El compilador le advierte cuando es necesario crear una copia defensiva. La propiedad `Distance` no cambia el estado, por lo que puede corregir esta advertencia si agrega el modificador `readonly` a la declaración:

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

Tenga en cuenta que el modificador `readonly` es necesario en una propiedad de solo lectura. El compilador no presupone que los descriptores de acceso `get` no modifican el estado; debe declarar `readonly` explícitamente. Las propiedades implementadas automáticamente son una excepción; el compilador tratará todos los captadores implementados automáticamente como `readonly`, por lo que aquí no es necesario agregar el modificador `readonly` a las propiedades `X` e `Y`.

El compilador aplica la regla por la que los miembros `readonly` no modifican el estado. El método siguiente no se compilará a menos que se quite el modificador `readonly`:

```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

Esta característica permite especificar la intención del diseño para que el compilador pueda aplicarla, y realizar optimizaciones basadas en dicha intención.

Para obtener más información, vea la sección Miembros de instancia `readonly` del artículo Tipos de estructura.

Métodos de interfaz predeterminados

Ahora puede agregar miembros a interfaces y proporcionar una implementación de esos miembros. Esta característica del lenguaje permite que los creadores de API agreguen métodos a una interfaz en versiones posteriores sin interrumpir la compatibilidad binaria o de origen con implementaciones existentes de dicha interfaz. Las implementaciones existentes *heredan* la implementación predeterminada. Esta característica también

permite que C# interopere con las API que tienen como destino Android o Swift, que admiten características similares. Los métodos de interfaz predeterminados también permiten escenarios similares a una característica del lenguaje de "rasgos".

Los métodos de interfaz predeterminados afectan a muchos escenarios y elementos del lenguaje. Nuestro primer tutorial abarca la actualización de una interfaz con implementaciones predeterminadas. Otros tutoriales y actualizaciones de referencia se incorporarán a tiempo en la versión general.

Más patrones en más lugares

La **coincidencia de patrones** ofrece herramientas que proporcionan funcionalidad dependiente de la forma entre tipos de datos relacionados pero diferentes. C# 7.0 introdujo la sintaxis de patrones de tipos y patrones de constantes mediante la expresión `is` y la instrucción `switch`. Estas características representaban los primeros pasos hacia el uso de paradigmas de programación donde los datos y la funcionalidad están separados. A medida que el sector se mueve hacia más microservicios y otras arquitecturas basadas en la nube, se necesitan otras herramientas de lenguaje.

C# 8.0 amplía este vocabulario para que pueda usar más expresiones de patrones en más lugares del código. Cuando los datos y la funcionalidad estén separados, tenga en cuenta estas características. Cuando los algoritmos dependan de un hecho distinto del tipo de entorno de ejecución de un objeto, tenga en cuenta la coincidencia de patrones. Estas técnicas ofrecen otra forma de expresar los diseños.

Además de nuevos patrones en nuevos lugares C# 8.0 agrega **patrones recursivos**. El resultado de cualquier expresión de patrón es una expresión. Un patrón recursivo es simplemente una expresión de patrón aplicada a la salida de otra expresión de patrón.

Expresiones switch

Con frecuencia, una instrucción `switch` genera un valor en cada uno de sus bloques `case`. Las **expresiones switch** le permiten usar una sintaxis de expresiones más concisa. Hay menos palabras clave `case` y `break` repetitivas y menos llaves. Por ejemplo, considere la siguiente enumeración que muestra los colores del arco iris:

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

Si la aplicación definió un tipo `RGBColor` construido a partir de los componentes `R`, `G` y `B`, podría convertir un valor `Rainbow` a sus valores RGB con el método siguiente que contiene una expresión switch:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid enum value", paramName:
nameof(colorBand)),
    };
```

Aquí hay varias mejoras en la sintaxis:

- La variable precede a la palabra clave `switch`. El orden diferente permite distinguir fácilmente la expresión switch de la instrucción switch.
- Los elementos `case` y `:` se reemplazan por `=>`. Es más concisa e intuitiva.
- El caso `default` se reemplaza por un descarte `_`.
- Los cuerpos son expresiones, no instrucciones.

Compárelo con el código equivalente mediante la instrucción clásica `switch`:

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

Patrones de propiedades

El **patrón de propiedades** permite hallar la coincidencia con las propiedades del objeto examinado. Considere un sitio de comercio electrónico que debe calcular los impuestos de ventas según la dirección del comprador. Ese cálculo no es una responsabilidad fundamental de una clase `Address`. Cambiará con el tiempo, probablemente con más frecuencia que el formato de dirección. El importe de los impuestos de ventas depende de la propiedad `State` de la dirección. El método siguiente usa el patrón de propiedades para calcular los impuestos de ventas a partir de la dirección y el precio:

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
```

La coincidencia de patrones crea una sintaxis concisa para expresar este algoritmo.

Patrones de tupla

Algunos algoritmos dependen de varias entradas. Los **patrones de tupla** permiten hacer cambios en función de varios valores, expresados como una tupla. El código siguiente muestra una expresión switch del juego *piedra, papel, tijeras*:

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```

Los mensajes indican el ganador. El caso de descarte representa las tres combinaciones de valores equivalentes, u otras entradas de texto.

Patrones posicionales

Algunos tipos incluyen un método `Deconstruct` que deconstruye sus propiedades en variables discretas. Cuando un método `Deconstruct` es accesible, puede usar **patrones posicionales** para inspeccionar las propiedades del objeto y usar esas propiedades en un patrón. Tenga en cuenta la siguiente clase `Point` que incluye un método `Deconstruct` con el objetivo de crear variables discretas para `X` y `Y`:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```

Además, tenga en cuenta la siguiente enumeración, que representa diversas posiciones de un cuadrante:

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

El método siguiente usa el **patrón posicional** para extraer los valores de `x` y `y`. A continuación, usa una cláusula `when` para determinar el valor `Quadrant` del punto:

```
static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};
```

El patrón de descarte del modificador anterior coincide cuando `x` o `y` es 0, pero no ambos. Una expresión switch debe generar un valor o producir una excepción. Si ninguno de los casos coincide, la expresión switch produce una excepción. El compilador genera una advertencia si no cubre todos los posibles casos en la expresión switch.

Puede explorar las técnicas de coincidencia de patrones en este tutorial avanzado sobre la coincidencia de patrones.

Declaraciones using

Una **declaración using** es una declaración de variable precedida por la palabra clave `using`. Indica al compilador que la variable que se declara debe eliminarse al final del ámbito de inclusión. Por ejemplo, considere el siguiente código que escribe un archivo de texto:

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    // Notice how we declare skippedLines after the using statement.
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}
```

En el ejemplo anterior, el archivo se elimina cuando se alcanza la llave de cierre del método. Ese es el final del

ámbito en el que se declara `file`. El código anterior es equivalente al siguiente código que usa las instrucciones using clásicas:

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    // We must declare the variable outside of the using block
    // so that it is in scope to be returned.
    int skippedLines = 0;
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
    } // file is disposed here
    return skippedLines;
}
```

En el ejemplo anterior, el archivo se elimina cuando se alcanza la llave de cierre asociada con la instrucción `using`.

En ambos casos, el compilador genera la llamada a `Dispose()`. El compilador genera un error si la expresión de la instrucción `using` no se puede eliminar.

Funciones locales estáticas

Ahora puede agregar el modificador `static` a funciones locales para asegurarse de que la función local no captura (hace referencia a) las variables del ámbito de inclusión. Si lo hace, se generará un error `CS8421` en el que se indicará que una función local estática no puede contener una referencia a <variable>.

Observe el código siguiente. La función local `LocalFunction` accede a la variable `y`, declarada en el ámbito de inclusión (el método `M`). Por lo tanto, `LocalFunction` no se puede declarar con el modificador `static`:

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

El código siguiente contiene una función local estática. Puede ser estática porque no accede a las variables del ámbito de inclusión:

```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

Estructuras ref descartables

Un elemento `struct` declarado con el modificador `ref` no puede implementar ninguna interfaz y, por tanto, no puede implementar IDisposable. Por lo tanto, para permitir que se elimine un elemento `ref struct`, debe tener un método `void Dispose()` accesible. Esta característica también se aplica a las declaraciones `readonly ref struct`.

Tipos de referencia que aceptan valores NULL

Dentro de un contexto de anotación que acepta valores NULL, cualquier variable de un tipo de referencia se considera un **tipo de referencia que no acepta valores NULL**. Si quiere indicar que una variable puede ser NULL, debe anexar `?` al nombre del tipo para declarar la variable como un **tipo de referencia que acepta valores NULL**.

En los tipos de referencia que no aceptan valores NULL, el compilador usa el análisis de flujo para garantizar que las variables locales se inicializan en un valor no NULL cuando se declaran. Los campos se deben inicializar durante la construcción. Si la variable no se establece mediante una llamada a alguno de los constructores disponibles o por medio de un inicializador, el compilador genera una advertencia. Además, los tipos de referencia que no aceptan valores NULL no pueden tener asignado un valor que podría ser NULL.

Los tipos de referencia que aceptan valores NULL no se comprueban para garantizar que se asignan o inicializan como NULL. Sin embargo, el compilador usa el análisis de flujo para garantizar que cualquier variable de un tipo de referencia que acepta valores NULL se compara con NULL antes de acceder a ella o de asignarse a un tipo de referencia que no los acepta.

Puede aprender más sobre la característica en la introducción a los tipos de referencia que aceptan valores NULL. Pruébela por su cuenta en una nueva aplicación en este tutorial de tipos de referencia que aceptan valores NULL. Puede encontrar más información sobre los pasos para migrar una base de código existente para hacer uso de los tipos de referencia que aceptan valores NULL en el tutorial sobre la migración de una aplicación para usar tipos de referencia que aceptan valores NULL.

Secuencias asincrónicas

A partir de C# 8.0, puede crear y consumir secuencias de forma asincrónica. Un método que devuelve una secuencia asincrónica tiene tres propiedades:

1. Se declara con el modificador `async`.
2. Devuelve IAsyncEnumerable<T>.
3. El método contiene instrucciones `yield return` que devuelven elementos sucesivos de la secuencia asincrónica.

Para consumir una secuencia asincrónica es necesario agregar la palabra clave `await` delante de la palabra clave `foreach` al enumerar los elementos de la secuencia. Para agregar la palabra clave `await` es necesario declarar el método que enumera la secuencia asincrónica con el modificador `async` y devolver un tipo permitido para un método `async`. Normalmente, esto significa devolver Task o Task<TResult>. También puede ser ValueTask o ValueTask<TResult>. Un método puede consumir y producir una secuencia asincrónica, lo que significa que devolvería IAsyncEnumerable<T>. El siguiente código genera una secuencia de 0 a 19, con una espera de 100 ms entre la generación de cada número:

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

Se enumeraría la secuencia mediante la instrucción `await foreach`:

```
await foreach (var number in GenerateSequence())
{
    Console.WriteLine(number);
}
```

Puede probar secuencias asincrónicas por su cuenta en nuestro tutorial sobre la creación y consumo de secuencias asincrónicas. Los elementos de secuencia se procesan de forma predeterminada en el contexto capturado. Si quiere deshabilitar la captura del contexto, use el método de extensión TaskAsyncEnumerableExtensions.ConfigureAwait. Para obtener más información sobre los contextos de sincronización y la captura del contexto actual, vea el artículo sobre el consumo del patrón asincrónico basado en tareas.

Asincrónica descartable

A partir C# 8.0, el lenguaje admite tipos descartables asincrónicos que implementan la interfaz System.IAsyncDisposable. El operando de una expresión `using` puede implementar IDisposable o IAsyncDisposable. En el caso de `IAsyncDisposable`, el compilador genera código para `await` el Task devuelto desde IAsyncDisposable.DisposeAsync. Para más información, consulte la instrucción `using`.

Índices y rangos

Los índices y rangos proporcionan una sintaxis concisa para acceder a elementos únicos o intervalos en una secuencia.

Esta compatibilidad con lenguajes se basa en dos nuevos tipos y dos nuevos operadores:

- System.Index representa un índice en una secuencia.
- El índice desde el operador final `^`, que especifica que un índice es relativo al final de la secuencia.
- System.Range representa un subrango de una secuencia.
- El operador de intervalo `..`, que especifica el inicio y el final de un intervalo como sus operandos.

Comencemos con las reglas de índices. Considere un elemento `sequence` de matriz. El índice `0` es igual que `sequence[0]`. El índice `^0` es igual que `sequence[sequence.Length]`. Tenga en cuenta que `sequence[^0]` produce una excepción, al igual que `sequence[sequence.Length]`. Para cualquier número `n`, el índice `^n` es igual que `sequence.Length - n`.

Un rango especifica el *inicio* y el *final* de un intervalo. El inicio del rango es inclusivo, pero su final es exclusivo, lo que significa que el *inicio* se incluye en el rango, pero el *final* no. El rango `[0..^0]` representa todo el intervalo, al igual que `[0..sequence.Length]` representa todo el intervalo.

Veamos algunos ejemplos. Tenga en cuenta la siguiente matriz, anotada con su índice desde el principio y desde el final:

```
var words = new string[]
{
                // index from start    index from end
    "The",      // 0                   ^9
    "quick",    // 1                   ^8
    "brown",    // 2                   ^7
    "fox",      // 3                   ^6
    "jumped",   // 4                   ^5
    "over",     // 5                   ^4
    "the",      // 6                   ^3
    "lazy",     // 7                   ^2
    "dog"       // 8                   ^1
};              // 9 (or words.Length) ^0
```

Puede recuperar la última palabra con el índice `^1`:

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

El siguiente código crea un subrango con las palabras "quick", "brown" y "fox". Va de `words[1]` a `words[3]`. El elemento `words[4]` no se encuentra en el intervalo.

```
var quickBrownFox = words[1..4];
```

El siguiente código crea un subrango con "lazy" y "dog". Incluye `words[^2]` y `words[^1]`. El índice del final `words[^0]` no se incluye:

```
var lazyDog = words[^2..^0];
```

En los ejemplos siguientes se crean rangos con final abierto para el inicio, el final o ambos:

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

También puede declarar rangos como variables:

```
Range phrase = 1..4;
```

El rango se puede usar luego dentro de los caracteres `[` y `]`:

```
var text = words[phrase];
```

No solo las matrices admiten índices y rangos. También puede usar índices y rangos con string, Span<T> o ReadOnlySpan<T>. Para más información, consulte Compatibilidad con tipos para los índices y los rangos.

Puede explorar más información acerca de los índices y los intervalos en el tutorial sobre índices e intervalos.

Asignación de uso combinado de NULL

C# 8.0 presenta el operador de asignación de uso combinado de `??=`. Puede usar el operador `??=` para asignar el valor de su operando derecho al operando izquierdo solo si el operando izquierdo se evalúa como `null`.

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers));  // output: 17 17
Console.WriteLine(i);  // output: 17
```

Para obtener más información, consulte Operadores ?? y ??.

Tipos construidos no administrados

En C# 7.3 y versiones anteriores, un tipo construido (es decir, un tipo que incluye al menos un argumento de tipo) no puede ser un tipo no administrado. A partir de C# 8.0, un tipo de valor construido no está administrado si solo contiene campos de tipos no administrados.

Por ejemplo, dada la siguiente definición del tipo genérico `Coords<T>`:

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

el tipo `Coords<int>` es un tipo no administrado en C# 8.0 y versiones posteriores. Al igual que en el caso de cualquier tipo no administrado, puede crear un puntero a una variable de este tipo o asignar un bloque de memoria en la pila para las instancias de este tipo:

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

Para más información, consulte Tipos no administrados.

Stackalloc en expresiones anidadas

A partir de C# 8.0, si el resultado de una expresión stackalloc es del tipo System.Span<T> o System.ReadOnlySpan<T>, puede usar la expresión `stackalloc` en otras expresiones:

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind);  // output: 1
```

Mejora de las cadenas textuales interpoladas

El orden de los tokens `$` y `@` en las cadenas textuales interpoladas puede ser cualquiera: tanto `$@"..."` como `@$"..."` son cadenas textuales interpoladas válidas. En versiones de C# anteriores, el token `$` debe aparecer delante del token `@`.

Novedades de C# 7.3

18/03/2020 • 13 minutes to read • Edit Online

Hay dos temas principales para la versión C# 7.3. Un tema proporciona características que permiten al código seguro ser tan eficaz como el código no seguro. El segundo tema proporciona mejoras incrementales en las características existentes. Además, se han agregado nuevas opciones de compilador en esta versión.

Las siguientes características nuevas admiten el tema del mejor rendimiento para código seguro:

- Puede tener acceso a campos fijos sin anclar.
- Puede volver a asignar variables locales `ref`.
- Puede usar inicializadores en matrices `stackalloc`.
- Puede usar instrucciones `fixed` con cualquier tipo que admita un patrón.
- Puede usar restricciones genéricas adicionales.

Se hicieron las mejoras siguientes a las características existentes:

- Puede probar `==` y `!=` con los tipos de tupla.
- Puede usar variables de expresión en más ubicaciones.
- Puede asociar atributos al campo de respaldo de las propiedades autoimplementadas.
- Se ha mejorado la resolución de métodos cuando los argumentos difieren por `in`.
- Ahora, la resolución de sobrecarga tiene menos casos ambiguos.

Las nuevas opciones del compilador son:

- `-publicsign` para habilitar la firma de ensamblados de software de código abierto (OSS).
- `-pathmap` para proporcionar una asignación para los directorios de origen.

El resto de este artículo proporciona información detallada y vínculos para obtener más información sobre cada una de las mejoras. Puede explorar estas características en su entorno mediante la herramienta global `dotnet try`:

1. Instale la herramienta global dotnet-try.
2. Clone el repositorio dotnet/try-samples.
3. Establezca el directorio actual en el subdirectorio *csharp7* para el repositorio *try-samples*.
4. Ejecute `dotnet try`.

Habilitación de código seguro más eficaz

Debería poder escribir código de C# de forma segura que tenga el mismo rendimiento que el código no seguro. El código seguro evita clases de errores, como desbordamientos de búfer, punteros perdidos y otros errores de acceso a la memoria. Estas nuevas características amplían las capacidades de código seguro comprobable. Procure escribir más código utilizando construcciones seguras. Estas características lo facilitan en gran medida.

Indexación de campos `fixed` sin requerir anclaje

Considere esta estructura:

```
unsafe struct S
{
    public fixed int myFixedField[10];
}
```

En versiones anteriores de C#, era necesario anclar una variable para acceder a uno de los enteros que forman parte de `myFixedField`. Ahora, el código siguiente se compila sin anclar la variable `p` dentro de una instrucción `fixed` independiente:

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        int p = s.myFixedField[5];
    }
}
```

La variable `p` tiene acceso a un elemento en `myFixedField`. No es necesario declarar otra variable `int*` independiente. Tenga en cuenta que todavía necesita un contexto `unsafe`. En versiones anteriores de C#, es necesario declarar un segundo puntero fijo:

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        fixed (int* ptr = s.myFixedField)
        {
            int p = ptr[5];
        }
    }
}
```

Para más información, consulte el artículo sobre la instrucción `fixed`.

Las variables locales de `ref` se pueden reasignar

Ahora, las variables locales de `ref` pueden reasignarse para hacer referencia a instancias diferentes después de haberse inicializado. El código siguiente ahora compila:

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Para obtener más información, vea el artículo sobre valores devueltos de `ref` y `ref` locales, así como el artículo sobre `foreach`.

Las matrices `stackalloc` admiten inicializadores

Ha podido especificar los valores para los elementos en una matriz cuando la inicializa:

```
var arr = new int[3] {1, 2, 3};
var arr2 = new int[] {1, 2, 3};
```

Ahora, esa misma sintaxis se puede aplicar a las matrices que se declaran con `stackalloc`:

```
int* pArr = stackalloc int[3] {1, 2, 3};
int* pArr2 = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc [] {1, 2, 3};
```

Para obtener más información, vea el artículo Operador `stackalloc`.

Hay más tipos compatibles con la instrucción `fixed`

La instrucción `fixed` admite un conjunto limitado de tipos. A partir de C# 7.3, cualquier tipo que contenga un método `GetPinnableReference()` que devuelve un `ref T` o `ref readonly T` puede ser `fixed`. La adición de esta característica significa que `fixed` puede utilizarse con System.Span<T> y tipos relacionados.

Para más información, vea el artículo sobre la instrucción `fixed` en la referencia del lenguaje.

Restricciones genéricas mejoradas

Ahora puede especificar el tipo System.Enum o System.Delegate como restricciones de clase base para un parámetro de tipo.

También puede usar la nueva restricción `unmanaged` para especificar que el parámetro de tipo debe ser un tipo no administrado que no acepta valores NULL.

Para obtener más información, vea los artículos sobre restricciones genéricas de `where` y restricciones sobre parámetros de tipo.

Agregar estas restricciones a tipos existentes es un cambio incompatible. Los tipos genéricos cerrados puede que ya no cumplan estas nuevas restricciones.

Mejora de las características existentes

El segundo tema proporciona las mejoras a las características del lenguaje. Estas características mejoran la productividad al escribir C#.

Compatibilidad con tuplas `==` y `!=`

Los tipos de tupla de C# ahora admiten `==` y `!=`. Para más información, vea la sección que trata la igualdad en el artículo sobre tuplas.

Asociación de atributos a los campos de respaldo para las propiedades autoimplementadas

Esta sintaxis ahora se admite:

```
[field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }
```

El atributo `SomeThingAboutFieldAttribute` se aplica al campo de respaldo generado por el compilador para `SomeProperty`. Para obtener más información, vea atributos en la guía de programación de C#.

Factor de desempate de resolución de sobrecargas del método `in`

Cuando se agregó el modificador de argumentos `in`, estos dos métodos causarían una ambigüedad:

```
static void M(S arg);
static void M(in S arg);
```

Ahora, la sobrecarga por valor (primera en el ejemplo anterior) es mejor que la de la versión de referencia de solo lectura. Para llamar a la versión con el argumento de referencia de solo lectura, debe incluir el modificador `in` cuando llame al método.

> **NOTE**
> Esto se implementó como una corrección de errores. Ya no es ambiguo, incluso con la versión de lenguaje establecida en "7.2".

Para obtener más información, consulte el artículo sobre el modificador de parámetros `in`.

Ampliación de variables de la expresión en inicializadores

La sintaxis que se agregó en C# 7.0 para permitir declaraciones de variable `out` se ha ampliado para incluir inicializadores de campo, inicializadores de propiedad, inicializadores de constructor y cláusulas de consulta. Permite código como el siguiente ejemplo:

```
public class B
{
   public B(int i, out int j)
   {
      j = i;
   }
}

public class D : B
{
   public D(int i) : base(i, out var j)
   {
      Console.WriteLine($"The value of 'j' is {j}");
   }
}
```

Mejoras en los candidatos de sobrecarga

En cada versión, las reglas de resolución de sobrecarga se actualizan para abordar situaciones en las que las invocaciones de métodos ambiguos tienen una opción "obvia". Esta versión agrega tres nuevas reglas para ayudar a que el compilador elija la opción obvia:

1. Cuando un grupo de métodos contiene tanto miembros de instancia como estáticos, el compilador descarta los miembros de la instancia si el método fue invocado sin un receptor o contexto de instancia. El compilador descarta los miembros estáticos si el método se invocó con un receptor de instancia. Cuando no hay ningún receptor, el compilador incluye solo miembros estáticos en un contexto estático; de lo contrario, miembros estáticos y de instancia. Cuando el receptor es ambiguamente una instancia o un tipo, el compilador incluye ambos. Un contexto estático, donde no se puede usar un receptor de instancia `this` implícito, incluye el cuerpo de miembros donde no se define `this`, como miembros estáticos, así como lugares donde `this` no se puede usar, como inicializadores de campo e inicializadores-constructores.
2. Cuando un grupo de métodos contiene algunos métodos genéricos cuyos argumentos de tipo no cumplen con sus restricciones, estos miembros se eliminan del conjunto de candidatos.
3. En el caso de una conversión de grupo de métodos, los métodos candidatos cuyo tipo de valor devuelto no coincide con el tipo de valor devuelto del delegado se eliminan del conjunto.

Solo notará este cambio porque encontrará menos errores de compilación para sobrecargas ambiguas de métodos cuando esté seguro de qué método es mejor.

Nuevas opciones del compilador

Las nuevas opciones del compilador admiten nuevos escenarios de DevOps y compilación de programas de C#.

Firma pública o de código abierto

La opción del compilador `-publicsign` indica al compilador que firme el ensamblado con una clave pública. El

ensamblado se marca como firmado, pero la firma se toma de la clave pública. Esta opción le permite crear ensamblados firmados a partir de proyectos de código abierto utilizando una clave pública.

Para obtener más información, vea el artículo -filealign (Opciones del compilador de C#).

pathmap

La opción del compilador `-pathmap` indica al compilador que reemplace las rutas de acceso de origen del entorno de compilación por rutas de acceso de origen asignadas. La opción `-pathmap` controla la ruta de acceso de origen escrita por el compilador en los archivos PDB o para CallerFilePathAttribute.

Para obtener más información, vea el artículo -pathmap (opciones del compilador de C#).

Novedades de C# 7.2

18/03/2020 • 7 minutes to read • Edit Online

C# 7.2 es otra versión de punto que agrega varias características útiles. Una de las ventajas de esta versión es que funciona mejor con tipos de valor, ya que evita copias o asignaciones innecesarias.

El resto de características son menos importantes, pero igualmente útiles.

C# 7.2 usa el elemento de configuración de selección de versión de lenguaje para seleccionar la versión del lenguaje del compilador.

Las nuevas características de lenguaje de esta versión son las siguientes:

- Técnicas para escribir código eficiente seguro
  - Una combinación de mejoras en la sintaxis que permiten trabajar con tipos de valor mediante la semántica de referencia.
- Argumentos con nombre no finales
  - Los argumentos con nombre pueden ir seguidos de argumentos posicionales.
- Caracteres de subrayado iniciales en literales numéricos
  - Los literales numéricos ahora pueden tener caracteres de subrayado iniciales antes de los dígitos impresos.
- Modificador de acceso `private protected`
  - El modificador de acceso `private protected` permite el acceso de clases derivadas en el mismo ensamblado.
- Expresiones `ref` condicionales
  - El resultado de una expresión condicional (`?:`) ahora puede ser una referencia.

En el resto de este artículo se proporciona información general sobre cada característica. Para cada característica, conocerá el razonamiento subyacente. Aprenderá la sintaxis. Puede explorar estas características en su entorno mediante la herramienta global `dotnet try`:

1. Instale la herramienta global dotnet-try.
2. Clone el repositorio dotnet/try-samples.
3. Establezca el directorio actual en el subdirectorio *csharp7* para el repositorio *try-samples*.
4. Ejecute `dotnet try`.

Mejoras de código eficiente seguro

Las características de lenguaje que presenta la versión 7.2 permiten trabajar con tipos de valor usando la semántica de referencia. Están diseñadas para aumentar el rendimiento minimizando la copia de tipos de valor sin usar las asignaciones de memoria asociadas al uso de tipos de referencia. Las características incluyen:

- El modificador `in` en los parámetros para especificar que un argumento se pasa mediante una referencia sin que el método al que se realiza una llamada lo modifique. Agregar el modificador `in` a un argumento es un cambio compatible con el origen.
- El modificador `ref readonly` en las devoluciones de método para indicar que un método devuelve su valor mediante una referencia, pero que no permite operaciones de escritura en el objeto. Agregar el modificador `ref readonly` es un cambio compatible con el origen, si el resultado devuelto se asigna a un valor. Agregar el modificador `readonly` a una instrucción return `ref` existente es un cambio incompatible. Requiere que los

autores de las llamadas actualicen la declaración de las variables locales `ref` para incluir el modificador `readonly`.

- La declaración `readonly struct` para indicar que una estructura es fija y que debería pasarse como parámetro `in` a los métodos de su miembro. Agregar el modificador `readonly` a una declaración struct existente es un cambio compatible con un elemento binario.
- La declaración `ref struct` para indicar que un tipo de estructura tiene acceso directo a la memoria administrada y que siempre debe estar asignada a la pila. Agregar el modificador `ref` a una declaración `struct` existente es un cambio incompatible. Un elemento `ref struct` no puede ser un miembro de una clase o usarse en otras ubicaciones donde puede asignarse en el montón.

Puede leer más información sobre todos estos cambios en Write safe efficient code (Escribir código eficiente seguro).

Argumentos con nombre no finales

Las llamadas de método ya pueden usar argumentos con nombre que precedan a argumentos posicionales si están en la posición adecuada. Para obtener más información, vea Argumentos opcionales y con nombre.

Caracteres de subrayado iniciales en literales numéricos

La implementación de la compatibilidad con separadores de dígitos en C# 7.0 no permitía que `_` fuera el primer carácter del valor literal. Los literales numéricos hexadecimales y binarios ya pueden empezar con un `_`.

Por ejemplo:

```
int binaryValue = 0b_0101_0101;
```

Modificador de acceso *private protected*

Presentamos un nuevo modificador de acceso compuesto: `private protected` indica que se puede tener acceso a un miembro mediante una clase o clases derivadas declaradas en un mismo ensamblado. Mientras que `protected internal` permite el acceso a las clases derivadas o clases que se encuentran en un mismo ensamblado, `private protected` limita el acceso a los tipos derivados que se declaran en un mismo ensamblado.

Para obtener más información, vea los modificadores de acceso en el material de referencia del lenguaje.

Expresiones `ref` condicionales

Por último, la expresión condicional puede producir un resultado de referencia en lugar de un resultado de valor. Por ejemplo, podría escribir lo siguiente para recuperar una referencia al primer elemento en una de dos matrices:

```
ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

La variable `r` es una referencia al primer valor de `arr` o `otherArr`.

Para más información, vea el operador condicional (?:) en la referencia del lenguaje.

Novedades de C# 7.1

18/03/2020 • 6 minutes to read • Edit Online

C# 7.1 es la primera versión secundaria del lenguaje C#. Marca una cadencia de versión mayor en el lenguaje. Podrá usar las nuevas características más pronto, lo ideal es que cuando vayan estando listas. C# 7.1 incluye la posibilidad de configurar el compilador para que coincida con una versión especificada del lenguaje. Ello permite aislar la decisión de actualizar las herramientas de la decisión de actualizar las versiones de lenguaje.

C# 7.1 incorpora el elemento de configuración de selección de versión de lenguaje, tres nuevas características de lenguaje y un nuevo comportamiento del compilador.

Las nuevas características de lenguaje de esta versión son las siguientes:

- Método `async` `Main`
  - El punto de entrada de una aplicación puede tener el modificador `async`.
- Expresiones literales `default`
  - Se pueden usar expresiones literales predeterminadas en expresiones de valor predeterminadas cuando el tipo de destino se pueda inferir.
- Nombres de elementos de tupla inferidos
  - En muchos casos, los nombres de elementos de tupla se pueden deducir de la inicialización de la tupla.
- Coincidencia de patrones en parámetros de tipo genérico
  - Puede usar expresiones de coincidencia de patrones en variables cuyo tipo es un parámetro de tipo genérico.

Por último, el compilador tiene dos opciones, `-refout` y `-refonly`, que controlan la generación de ensamblados de referencia.

Para usar las características más recientes en una versión secundaria, tendrá que configurar la versión del idioma de compilador y seleccionar la versión.

En el resto de este artículo se proporciona información general sobre cada característica. Para cada característica, conocerá el razonamiento subyacente. Aprenderá la sintaxis. Puede explorar estas características en su entorno mediante la herramienta global `dotnet try`:

1. Instale la herramienta global dotnet-try.
2. Clone el repositorio dotnet/try-samples.
3. Establezca el directorio actual en el subdirectorio *csharp7* para el repositorio *try-samples*.
4. Ejecute `dotnet try`.

Async main

Un método *async main* permite usar `await` en el método `Main`. Anteriormente, hubiera sido necesario escribir lo siguiente:

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

Ahora se puede escribir esto:

```
static async Task<int> Main()
{
    // This could also be replaced with the body
    // DoAsyncWork, including its await expressions:
    return await DoAsyncWork();
}
```

Si el programa no devuelve un código de salida, puede declarar un método `Main` que devuelva una Task:

```
static async Task Main()
{
    await SomeAsyncMethod();
}
```

En el artículo sobre async main de la guía de programación puede leer más detalles al respecto.

Expresiones literales predeterminadas

Las expresiones literales predeterminadas constituyen una mejora con respecto a las expresiones de valor predeterminadas. Estas expresiones inicializan una variable en el valor predeterminado. Donde anteriormente habría que escribir:

```
Func<string, bool> whereClause = default(Func<string, bool>);
```

Ahora, se puede pasar por alto el tipo del lado derecho de la inicialización:

```
Func<string, bool> whereClause = default;
```

Para más información, consulte la sección Literal default del artículo Operador default.

Nombres de elementos de tupla inferidos

Esta característica supone una pequeña mejora con respecto a la característica de tuplas incluida en C# 7.0. Muchas veces, cuando se inicializa una tupla, las variables usadas en el lado derecho de la asignación son las mismas que los nombres que querríamos dar a los elementos de tupla:

```
int count = 5;
string label = "Colors used in the map";
var pair = (count: count, label: label);
```

Ahora, los nombres de los elementos de tupla se pueden deducir de las variables empleadas para inicializar la tupla en C# 7.1:

```
int count = 5;
string label = "Colors used in the map";
var pair = (count, label); // element names are "count" and "label"
```

Encontrará más información sobre esta característica en el artículo sobre tuplas.

Coincidencia de patrones en parámetros de tipo genérico

A partir de C# 7.1, la expresión de patrón para `is` y el patrón de tipo `switch` pueden tener el tipo de un

parámetro de tipo genérico. Esto puede ser especialmente útil al comprobar los tipos que pueden ser tipos `struct` o `class` y si quiere evitar la conversión boxing.

Generación de ensamblados de referencia

Existen dos nuevas opciones del compilador con las que se generan *ensamblados solo de referencia*: -refout y -refonly. En los artículos de los vínculos se explican estas opciones y los ensamblados de referencia de manera más pormenorizada.

Novedades de C# 7.0

29/06/2020 • 27 minutes to read • Edit Online

C# 7.0 incorpora varias características nuevas al lenguaje C#:

- Variables de `out`
  - Puede declarar valores `out` insertados como argumentos en el método cuando se usen.
- Tuplas
  - Puede crear tipos ligeros sin nombre que contengan varios campos públicos. Los compiladores y las herramientas IDE comprenden la semántica de estos tipos.
- Descartes
  - Los descartes son variables temporales y de solo escritura que se usan en argumentos cuando el valor asignado es indiferente. Son especialmente útiles al deconstruir tuplas y tipos definidos por el usuario, así como al realizar llamadas a métodos con parámetros `out`.
- Coincidencia de patrones
  - Puede crear la lógica de bifurcación en función de tipos y valores arbitrarios de los miembros de esos tipos.
- Devoluciones y variables locales `ref`
  - Las variables locales de método y los valores devueltos pueden ser referencias a otro almacenamiento.
- Funciones locales
  - Puede anidar funciones en otras funciones para limitar su ámbito y visibilidad.
- Más miembros con forma de expresión
  - La lista de miembros que se pueden crear con expresiones ha crecido.
- Expresiones `throw`
  - Puede iniciar excepciones en construcciones de código que antes no se permitían porque `throw` era una instrucción.
- Tipos de valor devueltos de async generalizados
  - Los métodos declarados con el modificador `async` pueden devolver otros tipos además de `Task` y `Task<T>`.
- Mejoras en la sintaxis de literales numéricos
  - Nuevos tokens mejoran la legibilidad de las constantes numéricas.

En el resto de este artículo se proporciona información general sobre cada característica. Para cada característica, conocerá el razonamiento subyacente. Aprenderá la sintaxis. Puede explorar estas características en su entorno mediante la herramienta global `dotnet try`:

1. Instale la herramienta global dotnet-try.
2. Clone el repositorio dotnet/try-samples.
3. Establezca el directorio actual en el subdirectorio *csharp7* para el repositorio *try-samples*.
4. Ejecute `dotnet try`.

Variables `out`

En esta versión se ha mejorado la sintaxis existente que admite parámetros `out`. Ahora puede declarar variables `out` en la lista de argumentos de una llamada a método, en lugar de escribir una instrucción de declaración distinta:

```
if (int.TryParse(input, out int result))
    Console.WriteLine(result);
else
    Console.WriteLine("Could not parse input");
```

Para mayor claridad, puede que prefiera especificar el tipo de la variable `out`, tal y como se muestra anteriormente. Pero el lenguaje admite el uso de una variable local con tipo implícito:

```
if (int.TryParse(input, out var answer))
    Console.WriteLine(answer);
else
    Console.WriteLine("Could not parse input");
```

- El código es más fácil de leer.
  - Declare la variable out donde la use, no en otra línea anterior.
- No es preciso asignar ningún valor inicial.
  - Al declarar la variable `out` cuando se usa en una llamada de método, no podrá usarla accidentalmente antes de que se asigne.

Tuplas

C# ofrece una sintaxis enriquecida para clases y estructuras que se usa para explicar la intención del diseño. Pero a veces esa sintaxis enriquecida requiere trabajo adicional con apenas beneficio. Puede que a menudo escriba métodos que requieren una estructura simple que contenga más de un elemento de datos. Para admitir estos escenarios, se han agregado *tuplas* a C#. Las tuplas son estructuras de datos ligeros que contienen varios campos para representar los miembros de datos. Los campos no se validan y no se pueden definir métodos propios

> **NOTE**
> Las tuplas estaban disponibles antes de C# 7.0, pero no eran eficientes ni compatibles con ningún lenguaje. Esto significaba que solo se podía hacer referencia a los elementos tupla como `Item1`, `Item2`, por ejemplo. C# 7.0 presenta la compatibilidad de lenguaje con las tuplas, que permite usar nombres semánticos en los campos de una tupla mediante tipos de tupla nuevos y más eficientes.

Puede crear una tupla asignando un valor a cada miembro, y, opcionalmente, proporcionando nombres semánticos a cada uno de los miembros de la tupla:

```
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");
```

La tupla `namedLetters` contiene campos denominados `Alpha` y `Beta`. Esos nombres solo existen en tiempo de compilación y no se conservan, por ejemplo, al inspeccionar la tupla mediante la reflexión en tiempo de ejecución.

En la asignación de una tupla, también pueden especificarse los nombres de los campos a la derecha de la asignación:

```
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

Puede que a veces quiera desempaquetar los miembros de una tupla devueltos de un método. Para ello, declare distintas variables para cada uno de los valores de la tupla. Este desempaquetado se denomina *deconstrucción* de la tupla:

```
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

También puede proporcionar una deconstrucción similar para cualquier tipo de .NET. Un método `Deconstruct` se escribe como un miembro de la clase. Ese método `Deconstruct` proporciona un conjunto de argumentos `out` para cada una de las propiedades que quiere extraer. Tenga en cuenta que esta clase `Point` proporciona un método deconstructor que extrae las coordenadas `X` e `Y`:

```
public class Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y) =>
        (x, y) = (X, Y);
}
```

Puede extraer los campos individuales asignando un `Point` a una tupla:

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

Puede obtener más información sobre las tuplas en el artículo sobre tuplas.

Descartes

Habitualmente, al deconstruir una tupla o realizar una llamada a un método mediante parámetros `out`, debe definir una variable con un valor indiferente y que no vaya a usar. C# agrega la compatibilidad con *descartes* para gestionar este tipo de escenarios. Un descarte es una variable de solo escritura con el nombre `_` (el carácter de guion bajo). Puede asignar todos los valores que quiera descartar a una única variable. Un descarte se parece a una variable no asignada, aunque no puede usarse en el código (excepto la instrucción de asignación).

Los descartes se admiten en los escenarios siguientes:

- Al deconstruir tuplas o tipos definidos por el usuario.
- Al realizar llamadas a métodos mediante parámetros out.
- En una operación de coincidencia de patrones con las instrucciones is y switch.
- Como un identificador independiente cuando quiera identificar explícitamente el valor de una asignación como descarte.

En el ejemplo siguiente se define un método `QueryCityDataForYears` que devuelve una tupla de tipo 6 con datos de una ciudad correspondientes a dos años diferentes. La llamada de método del ejemplo se refiere únicamente a los dos valores de rellenado que devuelve el método, por lo que trata los valores restantes de la tupla como descartes al deconstruirla.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Para obtener más información, vea Descartes.

Detección de patrones

La *coincidencia de patrones* es una característica que permite implementar la distribución de métodos en propiedades distintas al tipo de un objeto. Probablemente ya esté familiarizado con la distribución de métodos en función del tipo de un objeto. En la programación orientada a objetos, los métodos virtuales y de invalidación proporcionan la sintaxis del lenguaje para implementar la distribución de métodos en función del tipo de un objeto. Las clases base y derivadas proporcionan distintas implementaciones. Las expresiones de coincidencia de patrones extienden este concepto para que se puedan implementar fácilmente patrones de distribución similares para tipos y elementos de datos que no se relacionan mediante una jerarquía de herencia.

La coincidencia de patrones admite expresiones `is` y `switch`. Cada una de ellas habilita la inspección de un objeto y sus propiedades para determinar si el objeto cumple el patrón buscado. Use la palabra clave `when` para especificar reglas adicionales para el patrón.

La expresión de patrón `is` extiende el conocido operador `is` para consultar el tipo de un objeto y asignar el resultado en una instrucción. En el código siguiente se comprueba si una variable es de tipo `int` y, si lo es, se agrega a la suma actual:

```
if (input is int count)
    sum += count;
```

En el pequeño ejemplo anterior se muestran las mejoras en la expresión `is`. Puede probar con tipos de valor y tipos de referencia, y asignar el resultado correcto a una nueva variable del tipo correcto.

La expresión de coincidencia switch tiene una sintaxis conocida, basada en la instrucción `switch` que ya forma parte del lenguaje C#. La instrucción switch actualizada tiene varias construcciones nuevas:

- El tipo de control de una expresión `switch` ya no se limita a tipos enteros, tipos `Enum`, `string` o a un tipo que acepta valores NULL correspondiente a uno de esos tipos. Se puede usar cualquier tipo.
- Puede probar el tipo de la expresión `switch` en todas las etiquetas `case`. Como sucede con la expresión `is`, puede asignar una variable nueva a ese tipo.
- Puede agregar una cláusula `when` para probar más condiciones en esa variable.
- Ahora el orden de las etiquetas `case` es importante. Se ejecuta la primera rama de la que se quiere obtener la coincidencia, mientras que el resto se omite.

En el código siguiente se muestran estas características:

```
public static int SumPositiveNumbers(IEnumerable<object> sequence)
{
    int sum = 0;
    foreach (var i in sequence)
    {
        switch (i)
        {
            case 0:
                break;
            case IEnumerable<int> childSequence:
            {
                foreach(var item in childSequence)
                    sum += (item > 0) ? item : 0;
                break;
            }
            case int n when n > 0:
                sum += n;
                break;
            case null:
                throw new NullReferenceException("Null found in sequence");
            default:
                throw new InvalidOperationException("Unrecognized type");
        }
    }
    return sum;
}
```

- `case 0:` es el patrón de constante conocido.
- `case IEnumerable<int> childSequence:` es un patrón de tipo.
- `case int n when n > 0:` es un patrón de tipo con una condición `when` adicional.
- `case null:` es el patrón NULL.
- `default:` es el caso predeterminado conocido.

Puede obtener más información sobre la coincidencia de patrones en Coincidencia de patrones en C#.

Devoluciones y variables locales ref

Esta característica habilita algoritmos que usan y devuelven referencias a variables definidas en otro lugar. Por ejemplo, trabajar con matrices de gran tamaño y buscar una sola ubicación con determinadas características. El método siguiente devuelve una **referencia** a ese almacenamiento en la matriz:

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

Puede declarar el valor devuelto como un elemento `ref` y modificar ese valor en la matriz, como se muestra en el código siguiente:

```
ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(item);
item = 24;
Console.WriteLine(matrix[4, 2]);
```

El lenguaje C# tiene varias reglas que impiden el uso incorrecto de las variables locales y devoluciones de `ref`:

- Tendrá que agregar la palabra clave `ref` a la firma del método y a todas las instrucciones `return` de un método.
  - Esto evidencia que el método se devuelve por referencia a lo largo del método.
- Se puede asignar `ref return` a una variable de valor, o bien a una variable `ref`.
  - El autor de la llamada controla si se copia el valor devuelto o no. La omisión del modificador `ref` al asignar el valor devuelto indica que el autor de la llamada quiere una copia del valor, no una referencia al almacenamiento.
- No se puede asignar un valor devuelto de método estándar a una variable local `ref`.
  - No permite instrucciones como `ref int i = sequence.Count();`
- No se puede devolver un elemento `ref` a una variable cuya duración se extiende más allá de la ejecución del método.
  - Esto significa que no se puede devolver una referencia a una variable local o a una variable con un ámbito similar.
- Las `ref` locales y las devoluciones no se pueden usar con métodos asíncronos.
  - El compilador no puede identificar si una variable a la que se hace referencia se ha establecido en su valor final en la devolución del método asíncrono.

La incorporación de variables locales ref y devoluciones de ref permite usar algoritmos que resultan más eficientes si se evita copiar los valores o se realizan operaciones de desreferencia varias veces.

Agregar `ref` al valor devuelto es un cambio compatible con el origen. El código existente se compila, pero el valor devuelto de referencia se copia cuando se asigna. Los autores de las llamadas deben actualizar el almacenamiento para el valor devuelto en una variable local `ref` para almacenar el valor devuelto como referencia.

Para más información, consulte el artículo sobre la palabra clave ref.

Funciones locales

Muchos diseños de clases incluyen métodos que se llaman desde una sola ubicación. Estos métodos privados adicionales mantienen cada método pequeño y centrado. Las *funciones locales* permiten declarar métodos en el contexto de otro método. Las funciones locales facilitan que los lectores de la clase vean que el método local solo se llama desde el contexto en el que se declara.

Hay dos casos de uso comunes para las funciones locales: métodos de iterador públicos y métodos asincrónicos públicos. Ambos tipos de métodos generan código que informa de errores más tarde de lo que los programadores

podrían esperar. En los métodos de iterador, las excepciones solo se observan al llamar a código que enumera la secuencia devuelta. En los métodos asincrónicos, las excepciones solo se observan cuando se espera al elemento `Task` devuelto. En el ejemplo siguiente se muestra la separación de la validación de parámetros de la implementación de iteradores mediante una función local:

```
public static IEnumerable<char> AlphabetSubset3(char start, char end)
{
    if (start < 'a' || start > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message: "start must be a letter");
    if (end < 'a' || end > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than {nameof(start)}");

    return alphabetSubsetImplementation();

    IEnumerable<char> alphabetSubsetImplementation()
    {
        for (var c = start; c < end; c++)
            yield return c;
    }
}
```

La misma técnica se puede emplear con métodos `async` para asegurarse de que las excepciones derivadas de la validación de argumentos se inician antes de comenzar el trabajo asincrónico:

```
public Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

NOTE

Algunos de los diseños que se admiten con funciones locales también se pueden realizar con *expresiones lambda*. Para más información, consulte Funciones locales frente a expresiones lambda.

Más miembros con forma de expresión

En C# 6 se presentaron los miembros con forma de expresión para funciones de miembros y propiedades de solo lectura. C# 7.0 amplía los miembros permitidos que pueden implementarse como expresiones. En C# 7.0, se pueden implementar *constructores*, *finalizadores* y descriptores de acceso `get` y `set` en *propiedades* e *indizadores*. En el código siguiente se muestran ejemplos de cada uno:

```
// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}
```

NOTE

En este ejemplo no se requiere un finalizador, pero se muestra para demostrar la sintaxis. No debe implementar un finalizador en la clase salvo que sea necesario para liberar recursos no administrados. También debe plantearse el uso de la clase SafeHandle en lugar de administrar directamente los recursos no administrados.

Estas nuevas ubicaciones para los miembros con forma de expresión representan un hito importante para el lenguaje C#: miembros de la comunidad que trabajan en el proyecto Roslyn de código abierto implementaron estas características.

Cambiar un método a un miembro con cuerpo de expresión es un cambio compatible con un elemento binario.

Expresiones throw

En C#, `throw` siempre ha sido una instrucción. Como `throw` es una instrucción, no una expresión, había construcciones de C# en las que no se podía usar. Incluyen expresiones condicionales, expresiones de fusión nulas y algunas expresiones lambda. La incorporación de miembros con forma de expresión agrega más ubicaciones donde las expresiones `throw` resultarían útiles. Para que pueda escribir cualquiera de estas construcciones, C# 7.0 presenta las *expresiones throw*.

Esta adición facilita la escritura de código más basado en expresiones. No se necesitan instrucciones adicionales para la comprobación de errores.

Tipos de valor devueltos de async generalizados

La devolución de un objeto `Task` desde métodos asincrónicos puede presentar cuellos de botella de rendimiento en determinadas rutas de acceso. `Task` es un tipo de referencia, por lo que su uso implica la asignación de un objeto. En los casos en los que un método declarado con el modificador `async` devuelva un resultado en caché o se complete sincrónicamente, las asignaciones adicionales pueden suponer un costo considerable de tiempo en secciones críticas para el rendimiento del código. Esas asignaciones pueden resultar costosas si se producen en bucles ajustados.

La nueva característica de lenguaje implica que los tipos de valor devuelto de métodos asincrónicos no están limitados a `Task`, `Task<T>` y `void`. El tipo devuelto debe seguir cumpliendo con el patrón asincrónico, lo que significa que debe haber un método `GetAwaiter` accesible. Como ejemplo concreto, se ha agregado el tipo `ValueTask` a .NET para sacar partido de esta nueva característica del lenguaje:

```
public async ValueTask<int> Func()
{
    await Task.Delay(100);
    return 5;
}
```

NOTE

Debe agregar el paquete de NuGet `System.Threading.Tasks.Extensions` para poder usar el tipo ValueTask<TResult>.

Esta mejora es especialmente útil para que los creadores de bibliotecas eviten la asignación de un elemento `Task` en código de rendimiento crítico.

Mejoras en la sintaxis de literales numéricos

La lectura incorrecta de constantes numéricas puede complicar la comprensión del código cuando se lee por primera vez. Las máscaras de bits u otros valores simbólicos son propensos a malentendidos. En C# 7.0 se incluyen dos nuevas características para escribir números de la manera más legible para el uso previsto: *literales binarios* y *separadores de dígitos*.

En aquellos casos en que cree máscaras de bits, o siempre que una representación binaria de un número aumente la legibilidad del código, escriba el número en formato binario:

```
public const int Sixteen =   0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

El `0b` al principio de la constante indica que el número está escrito como número binario. Los números binarios pueden ser muy largos, por lo que a menudo resulta más fácil ver los patrones de bits si se introduce `_` como separador de dígitos, como se ha mostrado antes en la constante binaria. El separador de dígitos puede aparecer en cualquier parte de la constante. En números de base 10, es habitual usarlo como separador de miles:

```
public const long BillionsAndBillions = 100_000_000_000;
```

El separador de dígitos también se puede usar con tipos `decimal`, `float` y `double`:

```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio = 1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

En conjunto, se pueden declarar constantes numéricas con mucha más legibilidad.

Novedades de C# 6

20/05/2020 • 18 minutes to read • Edit Online

La versión 6.0 de C# incluye muchas características que mejoran la productividad para los desarrolladores. El efecto general de estas características es que se escribe código más conciso y legible. La sintaxis contiene menos complejidad para muchas de las prácticas habituales. Es más fácil ver la intención del diseño con menos complejidad. Si conoce bien estas características, podrá mejorar la productividad y escribir código más legible. Puede concentrarse más en las características que en las construcciones del lenguaje.

El resto de este artículo proporciona información general de cada una de estas características, con un vínculo para examinar cada una. También puede examinar las características en una exploración interactiva sobre C# 6 en la sección de tutoriales.

Propiedades automáticas de solo lectura

Las propiedades automáticas de solo lectura proporcionan una sintaxis más concisa para crear tipos inmutables. La propiedad automática se declara solo con un descriptor de acceso get:

```
public string FirstName { get; }
public string LastName { get;  }
```

Las propiedades `FirstName` y `LastName` solo se pueden establecer en el cuerpo del constructor de la misma clase:

```
public Student(string firstName, string lastName)
{
    if (IsNullOrWhiteSpace(lastName))
        throw new ArgumentException(message: "Cannot be blank", paramName: nameof(lastName));
    FirstName = firstName;
    LastName = lastName;
}
```

Intentar establecer `LastName` en otro método genera un error de compilación `CS0200`:

```
public class Student
{
    public string LastName { get;  }

    public void ChangeName(string newLastName)
    {
        // Generates CS0200: Property or indexer cannot be assigned to -- it is read only
        LastName = newLastName;
    }
}
```

Esta característica habilita la compatibilidad de lenguaje real para crear tipos inmutables y usa la sintaxis de propiedades automáticas más concisa y cómoda.

Si al agregar esta sintaxis no se quita un método accesible, se trata de un cambio compatible con los elementos binarios.

Inicializadores de propiedades automáticas

Los *inicializadores de propiedades automáticas* permiten declarar el valor inicial de una propiedad automática como parte de la declaración de la propiedad.

```
public ICollection<double> Grades { get; } = new List<double>();
```

El miembro `Grades` se inicializa cuando se declara. Eso hace que sea más fácil realizar la inicialización exactamente una sola vez. La inicialización es parte de la declaración de la propiedad, lo que facilita igualar la asignación de almacenamiento con la interfaz pública para objetos `Student`.

Miembros de función con cuerpos de expresión

Muchos de los miembros que se escriben son instrucciones únicas que podrían ser expresiones únicas. Escriba un miembro con forma de expresión en su lugar. Funciona para métodos y propiedades de solo lectura. Por ejemplo, un reemplazo de `ToString()` suele ser un excelente candidato:

```
public override string ToString() => $"{LastName}, {FirstName}";
```

También puede usar esta sintaxis para propiedades de solo lectura:

```
public string FullName => $"{FirstName} {LastName}";
```

Cambiar un miembro existente por un miembro con cuerpo de expresión es un cambio compatible con un elemento binario.

uso de versión estática

La mejora *using static* permite importar los métodos estáticos de una sola clase. Se especifica la clase que se está usando:

```
using static System.Math;
```

Math no contiene ningún método de instancia. También se puede usar `using static` para importar los métodos estáticos de una clase para una clase que tiene métodos estáticos y de instancia. Uno de los ejemplos más útiles es String:

```
using static System.String;
```

> **NOTE**
> Se debe usar el nombre de clase completo, `System.String`, en una instrucción using estática. No se puede usar la palabra clave `string` en su lugar.

Al importar desde una instrucción `static using`, los métodos de extensión solo están en ámbito cuando se llaman mediante la sintaxis de invocación de métodos de extensión. No están en ámbito cuando se llaman como métodos estáticos. A menudo verá esto en las consultas LINQ. Puede importar el modelo LINQ mediante la importación de Enumerable o Queryable.

```
using static System.Linq.Enumerable;
```

Normalmente se llama a los métodos de extensión mediante expresiones de invocación de método de extensión. La ambigüedad se resuelve al agregar el nombre de clase en el caso excepcional en que se llaman mediante una llamada de método estático.

La directiva `static using` también importa cualquier tipo anidado. Es posible hacer referencia a los tipos anidados sin calificación.

Operadores condicionales NULL

El *operador condicional NULL* realiza comprobaciones NULL de una forma mucho más sencilla y fluida. Reemplace el acceso a miembros `.` por `?.`:

```
var first = person?.FirstName;
```

En el ejemplo anterior, se asigna `null` a la variable `first` si el objeto person es `null`. De lo contrario, se le asigna el valor de la propiedad `FirstName`. Y lo más importante es que `?.` significa que esta línea de código no genera una excepción `NullReferenceException` si la variable `person` es `null`. En su lugar, se cortocircuita y devuelve `null`. También puede usar un operador condicional NULL para el acceso de matriz o indizador. Reemplace `[]` por `?[]` en la expresión de índice.

La siguiente expresión devuelve `string`, independientemente del valor de `person`. A menudo se usa esta construcción con el operador de *fusión nula* para asignar valores predeterminados cuando una de las propiedades es `null`. Si la expresión se cortocircuita, se asigna tipo al valor `null` devuelto para que coincida con la expresión completa.

```
first = person?.FirstName ?? "Unspecified";
```

También se puede usar `?.` para invocar métodos de forma condicional. El uso más común de las funciones miembro con el operador condicional NULL es invocar de forma segura delegados (o controladores de eventos) que puedan ser `null`. Se llama al método `Invoke` del delegado con el operador `?.` para acceder al miembro. Puede ver un ejemplo en el artículo sobre patrones de delegado.

Las reglas del operador `?.` garantizan que el lado izquierdo del operador solo se evalúa una vez. Esto habilita muchos giros, incluido el ejemplo siguiente con controladores de eventos:

```
// preferred in C# 6:
this.SomethingHappened?.Invoke(this, eventArgs);
```

La garantía de que el lado izquierdo se evalúa solo una vez también permite usar cualquier expresión, incluidas llamadas a métodos, en el lado izquierdo de `?.`

Interpolación de cadenas

Con C# 6, la nueva característica de interpolación de cadenas permite insertar expresiones en una cadena. Solo hay que anteponer `$` a la cadena y usar expresiones entre `{` y `}` en lugar de ordinales:

```
public string FullName => $"{FirstName} {LastName}";
```

Este ejemplo usa propiedades para las expresiones sustituidas. Se puede usar cualquier expresión. Por ejemplo, se podría calcular la puntuación media de un alumno como parte de la interpolación:

```
public string GetGradePointPercentage() =>
    $"Name: {LastName}, {FirstName}. G.P.A: {Grades.Average():F2}";
```

La línea de código anterior dará formato al valor de `Grades.Average()` como un número de punto flotante con dos posiciones decimales.

A menudo puede ser necesario dar formato a la cadena generada con una referencia cultural concreta. Se aprovecha el hecho de que el objeto generado por una interpolación de cadena puede convertirse implícitamente a System.FormattableString. La instancia FormattableString contiene la cadena de formato compuesta y los resultados de la evaluación de las expresiones antes de convertirlas en cadenas. Use el método FormattableString.ToString(IFormatProvider) para especificar la referencia cultural cuando de formato a una cadena. En el ejemplo siguiente se genera una cadena con la referencia cultural de alemán (de-DE). (De forma predeterminada, la referencia cultural de alemán usa el carácter "," como separador de decimales y el carácter "." como separador de miles).

```
FormattableString str = $"Average grade is {s.Grades.Average()}";
var gradeStr = str.ToString(new System.Globalization.CultureInfo("de-DE"));
```

Para empezar a trabajar con la interpolación de cadenas, vea el tutorial interactivo Interpolación de cadenas en C#, el artículo Interpolación de cadenas y el tutorial Interpolación de cadenas en C#.

Filtros de excepciones

Los *filtros de excepciones* son cláusulas que determinan cuándo se debe aplicar una cláusula catch determinada. Si la expresión usada para un filtro de excepciones se evalúa como `true`, la cláusula catch realiza su procesamiento normal en una excepción. Si la expresión se evalúa como `false`, entonces se omite la cláusula `catch`. Un uso consiste en examinar la información sobre una excepción para determinar si una cláusula `catch` puede procesar la excepción:

```
public static async Task<string> MakeRequest()
{
    WebRequestHandler webRequestHandler = new WebRequestHandler();
    webRequestHandler.AllowAutoRedirect = false;
    using (HttpClient client = new HttpClient(webRequestHandler))
    {
        var stringTask = client.GetStringAsync("https://docs.microsoft.com/en-us/dotnet/about/");
        try
        {
            var responseText = await stringTask;
            return responseText;
        }
        catch (System.Net.Http.HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
    }
}
```

La expresión `nameof`

La expresión nameof se evalúa como el nombre de un símbolo. Es una excelente manera de hacer que las herramientas funcionen siempre que se necesita el nombre de una variable, una propiedad o un campo de miembro. Uno de los usos más comunes de `nameof` es para proporcionar el nombre de un símbolo que produjo una excepción:

```
if (IsNullOrWhiteSpace(lastName))
    throw new ArgumentException(message: "Cannot be blank", paramName: nameof(lastName));
```

Otro uso es con aplicaciones basadas en XAML que implementan la interfaz `INotifyPropertyChanged`:

```
public string LastName
{
    get { return lastName; }
    set
    {
        if (value != lastName)
        {
            lastName = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(LastName)));
        }
    }
}
private string lastName;
```

Await en bloques Catch y Finally

C# 5 tenía varias limitaciones en cuanto a dónde se podían colocar las expresiones `await`. Con C# 6, ahora se puede usar `await` en expresiones `catch` o `finally`. Se suele usar principalmente con escenarios de registro:

```
public static async Task<string> MakeRequestAndLogFailures()
{
    await logMethodEntrance();
    var client = new System.Net.Http.HttpClient();
    var streamTask = client.GetStringAsync("https://localHost:10000");
    try {
        var responseText = await streamTask;
        return responseText;
    } catch (System.Net.Http.HttpRequestException e) when (e.Message.Contains("301"))
    {
        await logError("Recovered from redirect", e);
        return "Site Moved";
    }
    finally
    {
        await logMethodExit();
        client.Dispose();
    }
}
```

Los detalles de implementación para agregar compatibilidad con `await` dentro de cláusulas `catch` y `finally` garantizan que el comportamiento sea coherente con el comportamiento del código sincrónico. Cuando el código que se ejecuta en una cláusula `catch` o `finally` produce una excepción, la ejecución busca una cláusula `catch` adecuada en el siguiente bloque adyacente. Si había una excepción actual, esa excepción se pierde. Lo mismo sucede con las expresiones de await en cláusulas `catch` y `finally`: se busca una cláusula `catch` adecuada y se pierde la excepción actual, si existe.

> **NOTE**
>
> Este comportamiento es el motivo por el que se recomienda escribir cláusulas `catch` y `finally` con cuidado, para evitar la introducción de nuevas excepciones.

Inicialización de colecciones asociativas mediante indexadores

Los *inicializadores de índice* son una de las dos características que hacen que los inicializadores de colección sean más coherentes con el uso del índice. En versiones anteriores de C#, se podían usar *inicializadores de colección* con colecciones de estilos de secuencia, incluido Dictionary<TKey,TValue>, al agregar llaves alrededor de los pares de clave y valor:

```
private Dictionary<int, string> messages = new Dictionary<int, string>
{
    { 404, "Page not Found"},
    { 302, "Page moved, but left a forwarding address."},
    { 500, "The web server can't come out to play today."}
};
```

Puede usarlos con colecciones Dictionary<TKey,TValue> y otros tipos donde el método `Add` accesible acepte más de un argumento. La nueva sintaxis admite la asignación con un índice en la colección:

```
private Dictionary<int, string> webErrors = new Dictionary<int, string>
{
    [404] = "Page not Found",
    [302] = "Page moved, but left a forwarding address.",
    [500] = "The web server can't come out to play today."
};
```

Esta característica significa que los contenedores asociativos se pueden inicializar mediante una sintaxis similar a la que se lleva usando en varias versiones para los contenedores de secuencia.

Métodos `Add` de extensión para inicializadores de colección

Otra característica que facilita la inicialización de colecciones es la capacidad de usar un *método de extensión* para el método `Add`. Esta característica se agregó por motivos de paridad con Visual Basic. La característica es más útil cuando se tiene una clase de colección personalizada que tiene un método con un nombre diferente para agregar nuevos elementos de forma semántica.

Mejoras en la resolución de sobrecarga

Es probable que esta última característica no se aprecie. Había construcciones en las que la versión anterior del compilador de C# podía considerar ambiguas algunas llamadas a métodos con expresiones lambda. Considere este método:

```
static Task DoThings()
{
     return Task.FromResult(0);
}
```

En versiones anteriores de C#, la llamada a este método con la sintaxis de grupo de método produciría un error:

```
Task.Run(DoThings);
```

El compilador anterior no podía distinguir correctamente entre `Task.Run(Action)` y `Task.Run(Func<Task>())`. En las versiones anteriores, era necesario usar una expresión lambda como argumento:

```
Task.Run(() => DoThings());
```

El compilador de C# 6 determina correctamente que `Task.Run(Func<Task>())` es una opción mejor.

Resultado del compilador determinista

La opción `-deterministic` indica al compilador que cree un ensamblado de salida idéntico byte a byte para las compilaciones sucesivas de los mismos archivos de código fuente.

De forma predeterminada, todas las compilaciones generan una salida única en cada compilación. El compilador agrega una marca de tiempo y genera un GUID a partir de números aleatorios. Use esta opción si quiere comparar la salida byte a byte para garantizar la coherencia entre las compilaciones.

Para obtener más información, vea el artículo -deterministic (opción del compilador).

Conozca los cambios más importantes en el compilador de C#.

18/03/2020 • 2 minutes to read • Edit Online

El equipo de Roslyn mantiene una lista de cambios importantes en los compiladores de C# y Visual Basic. Puede encontrar información sobre esos cambios en estos vínculos del repositorio de GitHub:

- Cambios importantes en VS2019 Update 1 y versiones posteriores en comparación con VS2019
- Cambios importantes desde VS2017 (C# 7)
- Cambios importantes en Roslyn 3.0 (VS2019) desde Roslyn 2.* (VS2017)
- Cambios importantes en Roslyn 2.0 (VS2017) desde Roslyn 1.* (VS2015) y en el compilador nativo de C# (VS2013 y anterior).
- Cambios importantes en Roslyn 1.0 (VS2015) desde el compilador nativo de C# (VS2013 y anterior).

Historia de C#

04/05/2020 • 23 minutes to read • Edit Online

En este artículo se proporciona un historial de cada versión principal del lenguaje C#. El equipo de C# continúa innovando y agregando nuevas características. Se puede encontrar información sobre el estado detallado de las características de lenguaje, incluidas las características consideradas para las próximas versiones, en el repositorio dotnet/roslyn de GitHub.

> **IMPORTANT**
>
> El lenguaje C# se basa en tipos y métodos en lo que la especificación de C# define como una *biblioteca estándar* para algunas de las características. La plataforma .NET ofrece los tipos y métodos en un número de paquetes. Un ejemplo es el procesamiento de excepciones. Cada expresión o instrucción `throw` se comprueba para asegurarse de que el objeto que se genera deriva de Exception. Del mismo modo, cada `catch` se comprueba para asegurarse de que el tipo que se captura deriva de Exception. Cada versión puede agregar requisitos nuevos. Para usar las características más recientes del lenguaje en entornos anteriores, es posible que tenga que instalar bibliotecas específicas. Estas dependencias están documentadas en la página de cada versión específica. Puede obtener más información sobre las relaciones entre lenguaje y biblioteca para tener más antecedentes sobre esta dependencia.

Las herramientas de compilación de C# consideran que la última versión principal del lenguaje es la versión predeterminada. Puede haber versiones secundarias entre versiones principales, que se detallan en los otros artículos de esta sección. Para usar las características más recientes en una versión secundaria, tendrá que configurar la versión del idioma de compilador y seleccionar la versión. Ha habido tres versiones secundarias desde C# 7.0:

- C# 7.3:
  - C# 7.3 está disponible a partir de la versión 15.7 de Visual Studio 2017 y el SDK de .NET Core 2.1.
- C# 7.2:
  - C# 7.2 está disponible a partir de la versión 15.5 de Visual Studio 2017 y el SDK de .NET Core 2.0.
- C# 7.1:
  - C# 7.1 está disponible a partir de la versión 15.3 de Visual Studio 2017 y el SDK de .NET Core 2.0.

C# versión 1.0

Si echa la vista atrás, la versión 1.0 de C#, publicada con Visual Studio .NET 2002, se parecía mucho a Java. Como parte de sus objetivos de diseño indicados para ECMA, intentaba ser un "lenguaje orientado a objetos que fuera sencillo, moderno y para fines generales". En aquel momento, parecerse a Java significaba que conseguía esos primeros objetivos de diseño.

Pero si volvemos a echarle un vistazo a C# 1.0 ahora, no lo verá tan claro. Carecía de capacidades asincrónicas integradas y de algunas funcionalidades útiles de genéricos que se dan por sentado. De hecho, carecía por completo de genéricos. ¿Y LINQ? Aún no estaba disponible. Esas características tardarían unos años más en agregarse.

C# 1.0 parecía estar privado de características, en comparación con la actualidad. Lo normal era tener que escribir código detallado. Pero aun así, hay que empezar por algo. C# 1.0 era una alternativa viable a Java en la plataforma Windows.

Las principales características de C# 1.0 incluían lo siguiente:

- Clases
- Structs
- Interfaces
- Eventos
- Propiedades
- Delegados
- Expresiones
- Instrucciones
- Atributos

Versión 1.2 de C#

Versión 1.2 de C# incluida en Visual Studio .NET 2003. Contenía algunas pequeñas mejoras del lenguaje. Lo más notable es que, a partir de esa versión, el código se generaba en un bucle `foreach` llamado Dispose en un IEnumerator cuando ese IEnumerator implementaba IDisposable.

C# versión 2.0

Aquí las cosas empiezan a ponerse interesantes. Echemos un vistazo a algunas de las principales características de C# 2.0, que se publicó en 2005 junto con Visual Studio 2005:

- Genéricos
- Tipos parciales
- Métodos anónimos
- Tipos de valores que aceptan valores NULL
- Iteradores
- Covarianza y contravarianza

Otras características de C# 2.0 agregaron capacidades a las características existentes:

- Accesibilidad independiente de captador o establecedor
- Conversiones de grupos de métodos (delegados)
- Clases estáticas
- Inferencia de delegados

Aunque puede que C# haya comenzado como un lenguaje genérico orientado a objetos, la versión 2.0 de C# cambió esto enseguida. En cuanto se pusieron con ella, se centraron en algunos puntos problemáticos graves para los desarrolladores. Y lo hicieron a lo grande.

Con los genéricos, los tipos y métodos pueden operar en un tipo arbitrario a la vez que conservan la seguridad de tipos. Por ejemplo, tener List<T> nos permite tener `List<string>` o `List<int>` y realizar operaciones de tipo seguro en esas cadenas o en enteros mientras los recorremos en iteración. Usar genéricos es mejor que crear un `ListInt` que derive de `ArrayList` o se convierta desde `Object` en cada operación.

C# 2.0 incorporó los iteradores. Para explicarlo brevemente, los iteradores permiten examinar todos los elementos de `List` (u otros tipos enumerables) con un bucle de `foreach`. Tener iteradores como una parte de primera clase del lenguaje mejoró drásticamente la facilidad de lectura del lenguaje y la capacidad de las personas de razonar sobre el código.

Aun así, C# seguía yendo por detrás de Java. Java ya había publicado versiones que incluían genéricos e iteradores. Pero esto cambiaría pronto a medida que los idiomas siguieran evolucionando.

C# versión 3.0

La versión 3.0 de C# llegó a finales de 2007, junto con Visual Studio 2008, aunque la cartera completa de características de lenguaje no llegaría realmente hasta la versión 3.5 de .NET Framework. Esta versión marcó un cambio importante en el crecimiento de C#. Estableció C# como un lenguaje de programación realmente formidable. Echemos un vistazo a algunas de las principales características de esta versión:

- Propiedades implementadas automáticamente
- Tipos anónimos (Guía de programación de C#).
- Expresiones de consulta
- Expresiones lambda
- Árboles de expresión
- Métodos de extensión
- Variables locales con asignación implícita de tipos
- Métodos parciales
- Inicializadores de objeto y colección

En retrospectiva, muchas de estas características parecen inevitables e indivisibles. Todas ellas encajan estratégicamente. Por lo general se considera que la mejor característica de la versión de C# fue la expresión de consulta, también conocida como Language-Integrated Query (LINQ).

Una vista más matizada examina árboles de expresión, expresiones lambda y tipos anónimos como la base sobre la que se construye LINQ. Sin embargo, en cualquier caso, C# 3.0 presentó un concepto revolucionario. C# 3.0 había comenzado a sentar las bases para convertir C# en un lenguaje híbrido funcional y orientado a objetos.

En concreto, permitía escribir consultas declarativas en estilo de SQL para realizar operaciones en colecciones, entre otras cosas. En lugar de escribir un bucle de `for` para calcular el promedio de una lista de enteros, permitía hacerlo fácilmente como `list.Average()`. La combinación de métodos de extensión y expresiones de consulta hizo que esa lista de enteros pareciera haberse vuelto más inteligente.

Llevó tiempo hasta que los usuarios realmente captaron e integraron el concepto, pero ocurrió gradualmente. Ahora, años más tarde, el código es mucho más conciso, sencillo y funcional.

C# versión 4.0

La versión 4.0 de C#, publicada con Visual Studio 2010, tuvo que lidiar con el carácter innovador que había adquirido la versión 3.0. Con la versión 3.0, el lenguaje de C# dejó de estar a la sombra de Java y alcanzó una posición prominente. El lenguaje se estaba convirtiendo rápidamente en algo elegante.

La siguiente versión introdujo algunas nuevas características interesantes:

- Enlace dinámico
- Argumentos opcionales/con nombre
- Covariante y contravariante de genéricos
- Tipos de interoperabilidad insertados

Los tipos de interoperabilidad insertados solucionaron un problema de implementación. La covarianza y contravarianza de genéricos proporcionan más capacidad para usar genéricos, pero son más bien académicos y probablemente más valorados por autores de bibliotecas y Framework. Los parámetros opcionales y con nombre permiten eliminar muchas sobrecargas de métodos y proporcionan mayor comodidad. Pero ninguna de esas características está modificando el paradigma exactamente.

La característica más importante fue la introducción de la palabra clave `dynamic`. Con la palabra clave `dynamic`, en la versión 4.0 de C# se introdujo la capacidad de invalidar el compilador durante la escritura en tiempo de compilación. Al usar la palabra clave dinámica, puede crear constructos similares a los lenguajes tipados dinámicamente, como JavaScript. Puede crear `dynamic x = "a string"` y luego agregarle seis, dejando que el

runtime decida qué debería suceder después.

Los enlaces dinámicos pueden dar lugar a errores, pero también otorgan un gran poder sobre el lenguaje.

C# versión 5.0

La versión 5.0 de C#, publicada con Visual Studio 2012, era una versión centrada del lenguaje. Casi todo el trabajo de esa versión se centró en otro concepto de lenguaje innovador: el modelo `async` y `await` para la programación asincrónica. Estas son las principales características:

- Miembros asincrónicos
- Atributos de información del llamador

Vea también

- Proyecto de código: Atributos de información del autor de llamada en C# 5.0

El atributo de información del autor de la llamada permite recuperar fácilmente información sobre el contexto donde se está ejecutando sin tener que recurrir a una gran cantidad de código de reflexión reutilizable. Tiene muchos usos en tareas de registro y diagnóstico.

Pero `async` y `await` son los auténticos protagonistas de esta versión. Cuando estas características salieron a la luz en 2012, C# cambió de nuevo las reglas del juego al integrar la asincronía en el lenguaje como un participante de primera clase. Si alguna vez ha trabajado con operaciones de larga duración y la implementación de sitios web de devoluciones de llamada, probablemente le haya encantado esta característica del lenguaje.

C# versión 6.0

Con las versiones 3.0 y 5.0, C# había agregado nuevas características destacables a un lenguaje orientado a objetos. Con la versión 6.0, publicada con Visual Studio 2015, en lugar de introducir una característica innovadora y predominante, se publicaron muchas características menores que aumentaron la productividad de la programación de C#. Estas son algunas de ellas:

- Importaciones estáticas
- Filtros de excepciones
- Inicializadores de propiedades automáticas
- Miembros de cuerpo de expresión
- Propagador de null
- Interpolación de cadenas
- operador nameof
- Inicializadores de índice

Entre las otras características nuevas se incluyen estas:

- Await en bloques catch y finally
- Valores predeterminados para las propiedades solo de captador

Cada una de estas características es interesante en sí misma. Pero si las observamos en su conjunto, vemos un patrón interesante. En esta versión, C# eliminó lenguaje reutilizable para que el código fuera más fluido y fácil de leer. Así que, para los que adoran el código simple y limpio, esta versión del lenguaje fue una gran aportación.

En esta versión también se hizo otra cosa, aunque no es una característica de lenguaje tradicional: publicaron el compilador Roslyn como un servicio. Ahora, el compilador de C# está escrito en C# y puede usarlo como parte de su trabajo de programación.

C# versión 7.0

C# versión 7.0 se comercializó con Visual Studio 2017. Esta versión tiene algunas cosas interesantes y evolutivas en la misma línea que C# 6.0, pero sin el compilador como servicio. Estas son algunas de las nuevas características:

- Variables out
- Tuplas y deconstrucción
- Coincidencia de patrones
- Funciones locales
- Miembros con forma de expresión expandidos
- Devoluciones y variables locales ref

Otras características incluidas:

- Descartes
- Literales binarios y separadores de dígitos
- Expresiones throw

Todas estas características ofrecen capacidades nuevas e interesantes para los desarrolladores y la posibilidad de escribir un código de manera más clara que nunca. De manera destacada, condensan la declaración de variables que se van a usar con la palabra clave `out` y permiten varios valores devueltos a través de tuplas.

Pero C# se está usando cada vez más. .NET Core ahora tiene como destino cualquier sistema operativo y tiene puesta la mirada en la nube y la portabilidad. Por supuesto, esas nuevas capacidades ocupan las ideas y el tiempo de los diseñadores de lenguaje, además de ofrecer nuevas características.

C# versión 7.1

C# empezó a publicar *versiones de punto* con C# 7.1. Esta versión agregó el elemento de configuración de selección de versión de lenguaje, tres nuevas características de lenguaje y un nuevo comportamiento del compilador.

Las nuevas características de lenguaje de esta versión son las siguientes:

- Método `async` `Main`
  - El punto de entrada de una aplicación puede tener el modificador `async`.
- Expresiones literales `default`
  - Se pueden usar expresiones literales predeterminadas en expresiones de valor predeterminadas cuando el tipo de destino se pueda inferir.
- Nombres de elementos de tupla inferidos
  - En muchos casos, los nombres de elementos de tupla se pueden deducir de la inicialización de la tupla.
- Coincidencia de patrones en parámetros de tipo genérico
  - Puede usar expresiones de coincidencia de patrones en variables cuyo tipo es un parámetro de tipo genérico.

Por último, el compilador tiene dos opciones, `-refout` y `-refonly`, que controlan la generación de ensamblados de referencia.

C# versión 7.2

C#7.2 agregó varias características de lenguaje pequeñas:

- Técnicas para escribir código eficiente seguro
  - Una combinación de mejoras en la sintaxis que permiten trabajar con tipos de valor mediante la semántica de referencia.
- Argumentos con nombre no finales

  - Los argumentos con nombre pueden ir seguidos de argumentos posicionales.
- Caracteres de subrayado iniciales en literales numéricos
  - Los literales numéricos ahora pueden tener caracteres de subrayado iniciales antes de los dígitos impresos.
- Modificador de acceso `private protected`
  - El modificador de acceso `private protected` permite el acceso de clases derivadas en el mismo ensamblado.
- Expresiones `ref` condicionales
  - El resultado de una expresión condicional (`?:`) ahora puede ser una referencia.

C# versión 7.3

Hay dos temas principales para la versión C# 7.3. Un tema proporciona características que permiten al código seguro ser tan eficaz como el código no seguro. El segundo tema proporciona mejoras incrementales en las características existentes. Además, se han agregado nuevas opciones de compilador en esta versión.

Las siguientes características nuevas admiten el tema del mejor rendimiento para código seguro:

- Puede acceder a campos fijos sin anclar.
- Puede reasignar variables locales `ref`.
- Puede usar inicializadores en matrices `stackalloc`.
- Puede usar instrucciones `fixed` con cualquier tipo que admita un patrón.
- Puede usar restricciones genéricas adicionales.

Se hicieron las mejoras siguientes a las características existentes:

- Puede probar `==` y `!=` con tipos de tupla.
- Puede usar variables de expresión en más ubicaciones.
- Puede asociar atributos al campo de respaldo de las propiedades autoimplementadas.
- Se ha mejorado la resolución de métodos cuando los argumentos difieren en `in`.
- La resolución de sobrecarga tiene ahora menos casos ambiguos.

Las nuevas opciones del compilador son:

- `-publicsign` para habilitar la firma de ensamblados de software de código abierto (OSS).
- `-pathmap` para proporcionar una asignación para los directorios de origen.

C# versión 8.0

C# 8.0 es la primera versión C# principal que tiene como destino específicamente .NET Core. Algunas características se basan en nuevas funcionalidades de CLR, otras en tipos de biblioteca agregados solo a .NET Core. C# 8.0 agrega las siguientes características y mejoras al lenguaje C#:

- Miembros de solo lectura
- Métodos de interfaz predeterminados
- Mejoras de coincidencia de patrones:
  - Expresiones switch
  - Patrones de propiedades
  - Patrones de tupla
  - Patrones posicionales
- Declaraciones using
- Funciones locales estáticas

- Estructuras ref descartables
- Tipos de referencia que aceptan valores null
- Secuencias asincrónicas
- Índices y rangos
- Asignación de uso combinado de NULL
- Tipos construidos no administrados
- Stackalloc en expresiones anidadas
- Mejora de las cadenas textuales interpoladas

Los miembros de interfaz predeterminados requieren mejoras en CLR. Estas características se agregaron en CLR para .NET Core 3.0. Los intervalos y los índices, y los flujos asincrónicos requieren nuevos tipos en las bibliotecas de .NET Core 3.0. Los tipos de referencia que aceptan valores NULL, aunque se implementan en el compilador, son mucho más útiles cuando se anotan bibliotecas para proporcionar información semántica relativa al estado NULL de los argumentos y los valores devueltos. Esas anotaciones se agregan a las bibliotecas de .NET Core.

Artículo publicado originalmente en el blog de NDepend , por cortesía de Erik Dietrich y Patrick Smacchia.

Relaciones entre características de lenguaje y tipos de biblioteca

18/03/2020 • 4 minutes to read • Edit Online

La definición del lenguaje C# exige que una biblioteca estándar tenga determinados tipos y determinados miembros accesibles en esos tipos. El compilador genera código que usa estos miembros y tipos necesarios para muchas características de lenguaje diferentes. En caso necesario, hay paquetes NuGet que contienen tipos necesarios para las versiones más recientes del lenguaje al escribir código para entornos donde esos tipos o miembros aún no se han implementado.

Esta dependencia de la funcionalidad de la biblioteca estándar ha formado parte del lenguaje C# desde su primera versión. En esa versión, los ejemplos incluían:

- Exception: usado para todas las excepciones generadas por el compilador.
- String: el tipo `string` de C# es sinónimo de String.
- Int32: sinónimo de `int`.

Esa primera versión era simple: el compilador y la biblioteca estándar se distribuían juntos y solo había una versión de cada uno.

Las versiones posteriores de C# a veces han agregado nuevos tipos o miembros a las dependencias. Los ejemplos incluyen: INotifyCompletion, CallerFilePathAttribute y CallerMemberNameAttribute. C# 7.0 continúa esta tendencia al agregar una dependencia a ValueTuple para implementar la característica de lenguaje tuplas.

El equipo de diseño del lenguaje se esfuerza por minimizar el área expuesta de los tipos y miembros necesarios en una biblioteca estándar compatible. Ese objetivo está equilibrado con un diseño limpio donde las nuevas características de la biblioteca se han incorporado sin problemas al lenguaje. Habrá nuevas características en versiones futuras de C# que exijan nuevos tipos y miembros en una biblioteca estándar. Es importante entender cómo administrar esas dependencias en el trabajo.

Administración de dependencias

Las herramientas del compilador de C# ahora se han desvinculado del ciclo de versiones de las bibliotecas de .NET en las plataformas compatibles. De hecho, las distintas bibliotecas de .NET tienen ciclos de versiones diferentes: .NET Framework en Windows se distribuye como una actualización de Windows, .NET Core se distribuye conforme a una programación independiente y las versiones de Xamarin de actualizaciones de la biblioteca se incluyen en las herramientas de Xamarin de cada plataforma de destino.

En la mayoría de las ocasiones estos cambios no son perceptibles. Pero cuando trabaje con una versión más reciente del lenguaje que requiera características aún no incluidas en las bibliotecas de .NET de esa plataforma, haga referencia a los paquetes NuGet para proporcionar esos nuevos tipos. A medida que las plataformas que admite la aplicación se actualicen con las nuevas instalaciones del marco de trabajo, puede quitar la referencia adicional.

Esta desvinculación significa que puede usar nuevas características de lenguaje aun cuando el destino sean equipos que no tengan el marco de trabajo correspondiente.

Consideraciones sobre versiones y actualizaciones para desarrolladores de C#

18/03/2020 • 4 minutes to read • Edit Online

La compatibilidad es un objetivo muy importante cuando se agregan nuevas características al lenguaje C#. En la mayoría de los casos, se puede volver a compilar el código existente con una nueva versión de compilador sin ningún problema.

Al adoptar nuevas características del lenguaje en una biblioteca, puede requerirse más atención. Puede crear una biblioteca con características que se encuentran en la última versión y necesita asegurarse de que las aplicaciones creadas con versiones anteriores del compilador pueden usarla. O bien puede actualizar una biblioteca existente, con la posibilidad de que muchos de los usuarios aún no tengan versiones actualizadas. Cuando tome decisiones sobre la adopción de nuevas características, se deberán tener en cuenta dos variaciones de compatibilidad: compatibilidad binaria y compatibilidad con el origen.

Cambios compatibles con elementos binarios

Los cambios en la biblioteca son **compatibles con elementos binarios** cuando se puede utilizar la biblioteca actualizada sin tener que volver a crear aplicaciones y bibliotecas que la usan. No es necesario volver a compilar ensamblados dependientes ni realizar ningún cambio en el código fuente. Los cambios compatibles con los elementos binarios también son compatibles con el origen.

Cambios compatibles con el origen

Los cambios en la biblioteca son **compatibles con el origen** cuando las aplicaciones y bibliotecas que usan la biblioteca no requieren cambios de código fuente, pero el origen debe volver a compilarse con la nueva versión para que funcione correctamente.

Cambios incompatibles

Si un cambio no es **compatible con el origen** ni es **compatible con los elementos binarios**, se necesitan cambios del código junto con la recompilación en las bibliotecas y aplicaciones dependientes.

Evaluar la biblioteca

Estos conceptos de compatibilidad afectan a las declaraciones públicas y protegidas de la biblioteca, no a su implementación interna. La adopción interna de nuevas características siempre es **compatible con los elementos binarios**.

Los cambios **compatibles con los elementos binarios** proporcionan la nueva sintaxis que genera el mismo código compilado para las declaraciones públicas que la sintaxis anterior. Por ejemplo, cambiar un método a un miembro con cuerpo de expresión es un **cambio compatible con un elemento binario**:

Código original:

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

Nuevo código:

```
public double CalculateSquare(double value) => value * value;
```

Los cambios **compatibles con el origen** presentan la sintaxis que cambia el código compilado para un miembro público, pero de una forma compatible con los sitios de llamada existentes. Por ejemplo, cambiar una firma del método a partir de un parámetro de valor a `in` mediante un parámetro de referencia es compatible con el origen, pero no lo es con los elementos binarios:

Código original:

```
public double CalculateSquare(double value) => value * value;
```

Nuevo código:

```
public double CalculateSquare(in double value) => value * value;
```

En el artículo de novedades, observe si la presentación de una característica que afecta a las declaraciones públicas es compatible con el origen o con los elementos binarios.

Tipos (Guía de programación de C#)

29/06/2020 • 23 minutes to read • Edit Online

Tipos, variables y valores

C# es un lenguaje fuertemente tipado. Todas las variables y constantes tienen un tipo, al igual que todas las expresiones que se evalúan como un valor. Cada declaración del método especifica un nombre, un número de parámetros, un tipo y una naturaleza (valor, referencia o salida) para cada parámetro de entrada y para el valor devuelto. La biblioteca de clases .NET define un conjunto de tipos numéricos integrados, así como tipos más complejos que representan una amplia variedad de construcciones lógicas, como el sistema de archivos, conexiones de red, colecciones y matrices de objetos, y fechas. Los programas de C# típicos usan tipos de la biblioteca de clases, así como tipos definidos por el usuario que modelan los conceptos que son específicos del dominio del problema del programa.

Entre la información almacenada en un tipo se puede incluir lo siguiente:

- El espacio de almacenamiento que requiere una variable del tipo.
- Los valores máximo y mínimo que puede representar.
- Los miembros (métodos, campos, eventos, etc.) que contiene.
- El tipo base del que hereda.
- La ubicación donde se asignará la memoria para variables en tiempo de ejecución.
- Los tipos de operaciones permitidas.

El compilador usa información de tipo para garantizar que todas las operaciones que se realizan en el código cuentan *con seguridad de tipos*. Por ejemplo, si declara una variable de tipo int, el compilador le permite usar la variable en operaciones de suma y resta. Si intenta realizar esas mismas operaciones en una variable de tipo bool, el compilador genera un error, como se muestra en el siguiente ejemplo:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
>
> Los desarrolladores de C y C++ deben tener en cuenta que, en C#, bool no se puede convertir en int.

El compilador inserta la información de tipo en el archivo ejecutable como metadatos. Common Language Runtime (CLR) usa esos metadatos en tiempo de ejecución para garantizar aún más la seguridad de tipos cuando asigna y reclama memoria.

Especificar tipos en declaraciones de variable

Cuando declare una variable o constante en un programa, debe especificar su tipo o usar la palabra clave var para que el compilador infiera el tipo. En el ejemplo siguiente se muestran algunas declaraciones de variable que utilizan tanto tipos numéricos integrados como tipos complejos definidos por el usuario:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Los tipos de parámetros del método y los valores devueltos se especifican en la declaración del método. En la siguiente firma se muestra un método que requiere una variable int como argumento de entrada y devuelve una cadena:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

Tras declarar una variable, no se puede volver a declarar con un nuevo tipo y no se le puede asignar un valor que no sea compatible con su tipo declarado. Por ejemplo, no puede declarar un valor int y, luego, asignarle un valor booleano de `true`. En cambio, los valores se pueden convertir en otros tipos, por ejemplo, cuando se asignan a variables nuevas o se pasan como argumentos de método. El compilador realiza automáticamente una *conversión de tipo* que no da lugar a una pérdida de datos. Una conversión que pueda dar lugar a la pérdida de datos requiere un valor *cast* en el código fuente.

Para obtener más información, vea Conversiones de tipos.

Tipos integrados

C# proporciona un conjunto estándar de tipos numéricos integrados para representar números enteros, valores de punto flotante, expresiones booleanas, caracteres de texto, valores decimales y otros tipos de datos. También hay tipos `string` y `object` integrados. Están disponibles para su uso en cualquier programa de C#. Para obtener una lista completa de los tipos integrados, vea Tipos integrados.

Tipos personalizados

Las construcciones struct, class, interface y enum se utilizan para crear sus propios tipos personalizados. La biblioteca de clases .NET es en sí misma una colección de tipos personalizados proporcionados por Microsoft que puede usar en sus propias aplicaciones. De forma predeterminada, los tipos usados con más frecuencia en la biblioteca de clases están disponibles en cualquier programa de C#. Otros están disponibles solo cuando agrega explícitamente una referencia de proyecto al ensamblado en el que se definen. Una vez que el compilador tenga una referencia al ensamblado, puede declarar variables (y constantes) de los tipos declarados en dicho ensamblado en el código fuente. Para más información, vea Biblioteca de clases .NET.

Common Type System

Es importante entender dos aspectos fundamentales sobre el sistema de tipos en .NET:

- Es compatible con el principio de herencia. Los tipos pueden derivarse de otros tipos, denominados *tipos base*. El tipo derivado hereda (con algunas restricciones), los métodos, las propiedades y otros miembros del tipo base. A su vez, el tipo base puede derivarse de algún otro tipo, en cuyo caso el tipo derivado hereda los miembros de ambos tipos base en su jerarquía de herencia. Todos los tipos, incluidos los tipos numéricos integrados como System.Int32 (palabra clave de C#: int), derivan en última instancia de un único tipo base, que es System.Object (palabra clave de C#: object). Esta jerarquía de tipos unificada se denomina Common Type System (CTS). Para más información sobre la herencia en C#, vea Herencia.
- En CTS, cada tipo se define como un *tipo de valor* o un *tipo de referencia*. Esto incluye todos los tipos personalizados de la biblioteca de clases .NET y también sus propios tipos definidos por el usuario. Los tipos que se definen mediante el uso de la palabra clave struct son tipos de valor; todos los tipos numéricos integrados son `structs`. Los tipos que se definen mediante el uso de la palabra clave class son tipos de referencia. Los tipos de referencia y los tipos de valor tienen distintas reglas de tiempo de compilación y distintos comportamientos de tiempo de ejecución.

En la ilustración siguiente se muestra la relación entre los tipos de valor y los tipos de referencia en CTS.

En la imagen siguiente se muestran tipos de valores y tipos de referencias en CTS:

> **NOTE**
> Puede ver que los tipos utilizados con mayor frecuencia están organizados en el espacio de nombres System. Sin embargo, el espacio de nombres que contiene un tipo no tiene ninguna relación con un tipo de valor o un tipo de referencia.

Tipos de valor

Los tipos de valor derivan de System.ValueType, el cual deriva de System.Object. Los tipos que derivan de System.ValueType tienen un comportamiento especial en CLR. Las variables de tipo de valor contienen directamente sus valores, lo que significa que la memoria se asigna insertada en cualquier contexto en el que se declare la variable. No se produce ninguna asignación del montón independiente ni sobrecarga de la recolección de elementos no utilizados para las variables de tipo de valor.

Existen dos categorías de tipos de valor: struct y enum.

Los tipos numéricos integrados son structs y tienen campos y métodos a los que se puede acceder:

```
// constant field on type byte.
byte b = byte.MaxValue;
```

Pero se declaran y se les asignan valores como si fueran tipos simples no agregados:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Los tipos de valor están *sellados*, lo que significa que, por ejemplo, no puede derivar un tipo de System.Int32, y no puede definir un struct para que herede de cualquier clase o struct definido por el usuario, porque un struct solo puede heredar de System.ValueType. A pesar de ello, un struct puede implementar una o más interfaces. Puede convertir un tipo struct en cualquier tipo de interfaz que implemente. Esto hace que una operación de conversión *boxing* encapsule el struct dentro de un objeto de tipo de referencia en el montón administrado. Las operaciones de conversión boxing se producen cuando se pasa un tipo de valor a un método que toma System.Object o cualquier tipo de interfaz como parámetro de entrada. Para obtener más información, vea Conversión boxing y unboxing.

Puede usar la palabra clave struct para crear sus propios tipos de valor personalizados. Normalmente, un struct se usa como un contenedor para un pequeño conjunto de variables relacionadas, como se muestra en el ejemplo siguiente:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

Para más información sobre estructuras, vea Tipos de estructura. Para más información sobre los tipos de valor, vea Tipos de valor.

La otra categoría de tipos de valor es enum. Una enumeración define un conjunto de constantes integrales con nombre. Por ejemplo, la enumeración System.IO.FileMode de la biblioteca de clases .NET contiene un conjunto de enteros constantes con nombre que especifican cómo se debe abrir un archivo. Se define como se muestra en el ejemplo siguiente:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

La constante `System.IO.FileMode.Create` tiene un valor de 2. Sin embargo, el nombre es mucho más significativo para los humanos que leen el código fuente y, por esa razón, es mejor utilizar enumeraciones en lugar de números literales constantes. Para obtener más información, vea System.IO.FileMode.

Todas las enumeraciones se heredan de System.Enum, el cual se hereda de System.ValueType. Todas las reglas que se aplican a las estructuras también se aplican a las enumeraciones. Para más información sobre las enumeraciones,

vea Tipos de enumeración.

Tipos de referencia

Un tipo que se define como clase, delegado, matriz o interfaz es un *tipo de referencia*. Al declarar una variable de un tipo de referencia en tiempo de ejecución, esta contendrá el valor null hasta que se cree explícitamente un objeto mediante el operador new, o bien que se le asigne un objeto creado en otro lugar mediante `new`, tal y como se muestra en el ejemplo siguiente:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

Una interfaz debe inicializarse junto con un objeto de clase que lo implementa. Si `MyClass` implementa `IMyInterface`, cree una instancia de `IMyInterface`, tal como se muestra en el ejemplo siguiente:

```
IMyInterface iface = new MyClass();
```

Cuando se crea el objeto, se asigna la memoria en el montón administrado y la variable solo contiene una referencia a la ubicación del objeto. Los tipos del montón administrado producen sobrecarga cuando se asignan y cuando los reclama la función de administración de memoria automática de CLR, conocida como *recolección de elementos no utilizados*. En cambio, la recolección de elementos no utilizados también está muy optimizada y no crea problemas de rendimiento en la mayoría de los escenarios. Para obtener más información sobre la recolección de elementos no utilizados, vea Administración de memoria automática.

Todas las matrices son tipos de referencia, incluso si sus elementos son tipos de valor. Las matrices derivan de manera implícita de la clase System.Array, pero el usuario las declara y las usa con la sintaxis simplificada que proporciona C#, como se muestra en el ejemplo siguiente:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Los tipos de referencia admiten la herencia completamente. Al crear una clase, puede heredar de cualquier otra interfaz o clase que no esté definida como sealed; y otras clases pueden heredar de la clase e invalidar sus métodos virtuales. Para más información sobre cómo crear clases propias, vea Clases y structs. Para más información sobre la herencia y los métodos virtuales, vea Herencia.

Tipos de valores literales

En C#, los valores literales reciben un tipo del compilador. Puede especificar cómo debe escribirse un literal numérico; para ello, anexe una letra al final del número. Por ejemplo, para especificar que el valor 4.56 debe tratarse como un valor flotante, anexe "f" o "F" después del número: `4.56f`. Si no se anexa ninguna letra, el compilador inferirá un tipo para el literal. Para obtener más información sobre los tipos que se pueden especificar con sufijos de letras, vea Tipos numéricos integrales y Tipos numéricos de punto flotante.

Dado que los literales tienen tipo y todos los tipos derivan en última instancia de System.Object, puede escribir y compilar código como el siguiente:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Tipos genéricos

Los tipos se pueden declarar con uno o varios *parámetros de tipo* que actúan como un marcador de posición para el tipo real (el *tipo concreto*) que proporcionará el código de cliente cuando cree una instancia del tipo. Estos tipos se denominan *tipos genéricos*. Por ejemplo, el tipo de .NET System.Collections.Generic.List<T> tiene un parámetro de tipo al que, por convención, se le denomina *T*. Cuando crea una instancia del tipo, especifica el tipo de los objetos que contendrá la lista, por ejemplo, la cadena:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

El uso del parámetro de tipo permite reutilizar la misma clase para incluir cualquier tipo de elemento, sin necesidad de convertir cada elemento en object. Las clases de colección genéricas se denominan *colecciones fuertemente tipadas* porque el compilador conoce el tipo específico de los elementos de la colección y puede generar un error en tiempo de compilación si, por ejemplo, intenta agregar un valor entero al objeto `stringList` del ejemplo anterior. Para más información, vea Genéricos.

Tipos implícitos, tipos anónimos y tipos que admiten un valor NULL

Como se ha mencionado anteriormente, puede asignar implícitamente un tipo a una variable local (pero no miembros de clase) mediante la palabra clave var. La variable sigue recibiendo un tipo en tiempo de compilación, pero este lo proporciona el compilador. Para más información, vea Variables locales con asignación implícita de tipos.

En algunos casos, resulta conveniente crear un tipo con nombre para conjuntos sencillos de valores relacionados que no desea almacenar ni pasar fuera de los límites del método. Puede crear *tipos anónimos* para este fin. Para obtener más información, consulte Tipos anónimos (Guía de programación de C#).

Los tipos de valor normales no pueden tener un valor null, pero se pueden crear tipos de valor que aceptan valores NULL mediante la adición de `?` después del tipo. Por ejemplo, `int?` es un tipo `int` que también puede tener el valor null. Los tipos que admiten un valor NULL son instancias del tipo struct genérico System.Nullable<T>. Los tipos que admiten un valor NULL son especialmente útiles cuando hay un intercambio de datos con bases de datos en las que los valores numéricos podrían ser nulos. Para más información, vea Tipos que admiten un valor NULL.

Secciones relacionadas

Para obtener más información, vea los temas siguientes:

- Conversiones de tipos
- Conversión boxing y conversión unboxing
- Uso de tipo dinámico
- Tipos de valor

- Tipos de referencia
- Clases y structs
- Tipos anónimos
- Genéricos

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Conversión de tipos de datos XML
- Tipos enteros

Tipos de referencia que aceptan valores NULL

29/06/2020 • 16 minutes to read • Edit Online

Una de las novedades de C# 8.0 son los **tipos de referencia que aceptan valores NULL** y los **tipos de referencia que no aceptan valores NULL**, que permiten usar instrucciones importantes sobre las propiedades de variables de tipos de referencia:

- **Se supone que una referencia no debe ser NULL**. Si las variables no deben ser NULL, el compilador aplica reglas que garantizan que sea seguro desreferenciar dichas variables sin comprobar primero que no se trata de un valor NULL:
  - La variable debe inicializarse como un valor distinto a NULL.
  - No se puede asignar el valor `null` a la variable.
- **Una referencia puede ser NULL**. Si las variables pueden ser NULL, el compilador aplica reglas para garantizar que haya comprobado correctamente si hay referencias NULL:
  - Solo se puede desreferenciar la variable si el compilador pueda garantizar que el valor no sea NULL.
  - Estas variables se pueden inicializar con el valor `null` predeterminado, y se les puede asignar el valor `null` en otro código.

Esta nueva característica proporciona grandes ventajas sobre el control de variables de referencia con respecto a versiones anteriores de C#, donde la intención del diseño no se puede determinar a partir de la declaración de la variable. El compilador no proporcionaba protección contra excepciones de referencia NULL para los tipos de referencia:

- **Una referencia puede ser NULL**. El compilador no emite ninguna advertencia cuando un tipo de referencia se inicializa con un valor NULL o posteriormente se asigna a un valor NULL. El compilador emite advertencias cuando estas variables se desreferencian sin comprobaciones de valores NULL.
- **Se asume que una referencia no es NULL**. Si se desreferencian tipos de referencia, el compilador no emite ninguna advertencia. El compilador emite advertencias si una variable se establece en una expresión que puede ser NULL.

Estas advertencias se emiten en tiempo de compilación. El compilador no agrega comprobaciones de valores NULL ni otras construcciones de tiempo de ejecución en un contexto que admite un valor NULL. En tiempo de ejecución, una referencia que acepta valores NULL y una referencia que no acepta valores NULL son equivalentes.

Con la incorporación de los tipos de referencia que aceptan valores NULL, puede declarar su intención de forma más clara. El valor `null` es la forma adecuada de representar que una variable que no hace referencia a un valor. No use esta característica para eliminar todos los valores `null` de su código. En su lugar, debería declarar su intención al compilador para que los demás desarrolladores la puedan ver al leer el código. Al declarar su intención, el compilador le informa de cuándo escribe código que no es coherente con esa intención.

Un **tipo de referencia que acepta valores NULL** se anota con la misma sintaxis que los tipos de valor que aceptan valores NULL: se agrega `?` junto al tipo de la variable. Por ejemplo, la siguiente declaración de variable representa una variable de cadena que acepta valores NULL, `name`:

```
string? name;
```

Cualquier variable en la que `?` no se anexe al nombre de tipo es un **tipo de referencia que no acepta valores NULL**. Esto incluye todas las variables de tipo de referencia en el código existente en el momento en el que se habilita esta característica.

El compilador usa el análisis estático para determinar si se sabe si una referencia que acepta valores NULL no tiene este tipo de valor. Si desreferencia una referencia que acepta valores NULL cuando esta puede ser NULL, el compilador genera una advertencia. Puede invalidar este comportamiento con el operador de limitación de advertencias de valores NULL `!` después del nombre de una variable. Por ejemplo, si sabe que la variable `name` no es NULL, pero el compilador genera una advertencia, puede escribir el código siguiente para invalidar el análisis del compilador:

```
name!.Length;
```

Nulabilidad de tipos

Cualquier tipo de referencia puede tener una de cuatro *nulabilidades*, que describen cuándo se generan las advertencias:

- *No acepta valores NULL*: no se pueden asignar valores NULL a las variables de este tipo. No es necesario comprobar si estas tienen un valor NULL antes de desreferenciarlas.
- *Acepta valores NULL*: se pueden asignar valores NULL a las variables de este tipo. Si se desreferencian sin comprobar primero la existencia de valores `null`, se producirá una advertencia.
- *Inconsciente*: se trata del estado previo a C# 8.0. Las variables de este tipo se pueden desreferenciar o asignar sin advertencias.
- *Desconocido*: este es generalmente el caso de los parámetros de tipo en los que las restricciones no indican al compilador que el tipo debe *aceptar valores NULL* o *no aceptar valores NULL*.

La nulabilidad de un tipo en una declaración de variable se controla mediante el *contexto que acepta valores NULL* en el que se declara la variable.

Contextos que aceptan valores NULL

Los contextos que aceptan valores NULL permiten un control preciso sobre cómo interpreta el compilador las variables de tipos de referencia. El **contexto de anotación que acepta valores NULL** de cualquier línea de origen está habilitado o deshabilitado. Puede considerar que el compilador anterior al de C# 8.0 compilaba todo el código con el contexto que acepta valores NULL deshabilitado: cualquier tipo de referencia puede tener el valor NULL. El **contexto de advertencia que acepta valores NULL** también se puede habilitar o deshabilitar. Este contexto especifica las advertencias que genera el compilador usando el análisis de flujos.

Tanto el contexto de anotación que acepta valores NULL como el contexto de advertencia que acepta valores NULL pueden establecerse para un proyecto con el elemento `Nullable` del archivo *.csproj*. Este elemento configura la forma en la que el compilador interpreta la nulabilidad de los tipos y las advertencias que se generan. Estos son los valores válidos:

- `enable`: el contexto de anotación que acepta valores NULL es **enabled**. El contexto de advertencia que acepta valores NULL es **enabled**.
  - Las variables de un tipo de referencia, como `string`, no aceptan valores NULL. Todas las advertencias de nulabilidad están habilitadas.
- `warnings`: el contexto de anotación que acepta valores NULL es **disabled**. El contexto de advertencia que acepta valores NULL es **enabled**.
  - Las variables de un tipo de referencia son inconscientes. Todas las advertencias de nulabilidad están habilitadas.
- `annotations`: el contexto de anotación que acepta valores NULL es **enabled**. el contexto de advertencia que acepta valores NULL es **disabled**.
  - Las variables de un tipo de referencia, como cadena, no admiten un valor NULL. Todas las advertencias de nulabilidad están deshabilitadas.

- `disable`: el contexto de anotación que acepta valores NULL es **disabled**. el contexto de advertencia que acepta valores NULL es **disabled**.
  - Las variables de tipo de referencia son inconscientes, como en versiones anteriores de C#. Todas las advertencias de nulabilidad están deshabilitadas.

Ejemplo:

```
<Nullable>enable</Nullable>
```

También puede usar directivas para establecer los mismos contextos en cualquier lugar del proyecto:

- `#nullable enable`: establece el contexto de anotación que acepta valores NULL y el contexto de advertencia que acepta valores NULL en **enabled**.
- `#nullable disable`: establece el contexto de anotación que acepta valores NULL y el contexto de advertencia que acepta valores NULL en **disabled**.
- `#nullable restore`: restaura el contexto de anotación que acepta valores NULL y el contexto de advertencia que acepta valores NULL según la configuración del proyecto.
- `#nullable disable warnings`: establece el contexto de advertencia que acepta valores NULL en **disabled**.
- `#nullable enable warnings`: establece el contexto de advertencia que acepta valores NULL en **enabled**.
- `#nullable restore warnings`: restaura el contexto de advertencia que acepta valores NULL según la configuración del proyecto.
- `#nullable disable annotations`: establezca el contexto de anotación que admite un valor NULL en **disabled**.
- `#nullable enable annotations`: establezca el contexto de anotación que admite un valor NULL en **enabled**.
- `#nullable restore annotations`: restaura el contexto de advertencia de anotación según la configuración del proyecto.

De forma predeterminada, los contextos de advertencias y anotaciones que aceptan valores NULL están **deshabilitados**. Esto implica que el código existente se compila sin cambios y sin generar ninguna advertencia nueva.

Estas opciones proporcionan dos estrategias distintas para actualizar un código base existente para usar tipos de referencia que aceptan valores NULL.

Contexto de anotación que admite valores NULL

El compilador usa las siguientes reglas en un contexto de anotación deshabilitado que acepta valores NULL:

- No se pueden declarar referencias que aceptan valores NULL en un contexto deshabilitado.
- Todas las variables de referencia se pueden asignar a un valor NULL.
- No se genera ninguna advertencia al desreferenciar una variable de un tipo de referencia.
- El operador de limitación de advertencias de valores NULL no se puede usar en un contexto deshabilitado.

El comportamiento es el mismo que en versiones anteriores de C#.

El compilador usa las siguientes reglas en un contexto de anotación habilitado que acepta valores NULL:

- Todas las variables de tipo de referencia son **referencias que no aceptan valores NULL**.
- Estas referencias se pueden desreferenciar sin problemas.
- Todos los tipos de referencia que aceptan valores NULL (con la anotación de `?` después del tipo en la declaración de la variable) pueden ser NULL. El análisis estático determina si se sabe que el valor no admite valores NULL cuando se desreferencia. Si no, el compilador emite una advertencia.
- Puede usar el operador de limitación de advertencias de valores NULL para declarar que una referencia que acepta valores NULL no tiene un valor NULL.

En un contexto de anotación que acepta valores NULL, el carácter `?` junto a un tipo de referencia declara un **tipo de referencia que acepta valores NULL**. Se puede incorporar un **operador de limitación de advertencias de valores NULL** `!` junto a una expresión para declarar que no tiene un valor NULL.

Contexto de advertencia que acepta valores NULL

El contexto de advertencia que acepta valores NULL no es igual al contexto de anotación que acepta valores NULL. Se pueden habilitar las advertencias incluso aunque las nuevas anotaciones estén deshabilitadas. El compilador usa el análisis de flujos estático para determinar el **estado NULL** de cualquier referencia. El estado NULL puede ser **no NULL** o **quizás NULL** cuando el *contexto de advertencia que acepta valores NULL* no está **deshabilitado**. Si desreferencia una referencia cuando el compilador ha determinado el estado **quizás NULL**, el compilador emite una advertencia. El estado de una referencia es **quizás NULL** a menos que el compilador pueda determinar una de estas dos condiciones:

1. La variable se ha asignado definitivamente a un valor distinto a NULL.
2. Se ha comprobado que la variable o la expresión no tiene un valor NULL antes de desreferenciarla.

El compilador genera advertencias cuando se desreferencia una variable o expresión que tiene un estado **quizás NULL** en un contexto de advertencia que admite valores NULL. Además, el compilador genera advertencias cuando se asigna una variable o expresión **quizás NULL** a un tipo de referencia que no acepta valores NULL en un contexto de anotación que sí los acepta.

Atributos que describen las API

Agregue atributos a las API que proporcionen al compilador más información sobre cuándo los argumentos o valores devueltos pueden admitir o no valores NULL. Puede obtener más información sobre estos atributos en nuestro artículo de la referencia del lenguaje que trata de los atributos que admiten valores NULL. Estos atributos se agregan a las bibliotecas de .NET a través de las versiones actuales y futuras. Las API que se usan más a menudo se actualizan primero.

Vea también

- Borrador de especificación de tipos de referencia que aceptan valores NULL
- Tutorial de introducción a las referencias que no aceptan valores NULL
- Migración de un código base existente a referencias que aceptan valores NULL
- -nullable (opción del compilador de C#)

Espacios de nombres (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Los espacios de nombres se usan mucho en programación de C# de dos maneras. En primer lugar, .NET usa espacios de nombres para organizar sus clases de la siguiente manera:

```
System.Console.WriteLine("Hello World!");
```

System es un espacio de nombres y Console es una clase de ese espacio de nombres. La palabra clave `using` se puede usar para que no se necesite el nombre completo, como en el ejemplo siguiente:

```
using System;
```

```
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

Para más información, vea using (Directiva).

En segundo lugar, declarar sus propios espacios de nombres puede ayudarle a controlar el ámbito de nombres de clase y método en proyectos de programación grandes. Use la palabra clave namespace para declarar un espacio de nombres, como en el ejemplo siguiente:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

El nombre del espacio de nombres debe ser un nombre de identificador de C# válido.

Información general sobre los espacios de nombres

Los espacios de nombres tienen las propiedades siguientes:

- Organizan proyectos de código de gran tamaño.
- Se delimitan mediante el operador `.`.
- La directiva `using` obvia la necesidad de especificar el nombre del espacio de nombres para cada clase.
- El espacio de nombres `global` es el espacio de nombres "raíz": `global::System` siempre hará referencia al espacio de nombres System de .NET.

especificación del lenguaje C#

Para más información, vea la sección Espacio de nombres de la Especificación del lenguaje C#.

Vea también

- Guía de programación de C#
- Utilizar espacios de nombres
- Procedimiento para usar el espacio de nombres My
- Nombres de identificador
- using (directiva)
- Operador ::

Tipos, variables y valores

29/06/2020 • 12 minutes to read • Edit Online

C# es un lenguaje fuertemente tipado. Todas las variables y constantes tienen un tipo, al igual que todas las expresiones que se evalúan como un valor. Cada una de las firmas de método especifica un tipo para cada parámetro de entrada y para el valor devuelto. La biblioteca de clases .NET Framework define un conjunto de tipos numéricos integrados, así como tipos más complejos que representan una amplia variedad de construcciones lógicas, como el sistema de archivos, conexiones de red, colecciones y matrices de objetos, y fechas. Los programas de C# típicos usan tipos de la biblioteca de clases, así como tipos definidos por el usuario que modelan los conceptos que son específicos del dominio del problema del programa.

Entre la información almacenada en un tipo se puede incluir lo siguiente:

- El espacio de almacenamiento que requiere una variable del tipo.
- Los valores máximo y mínimo que puede representar.
- Los miembros (métodos, campos, eventos, etc.) que contiene.
- El tipo base del que hereda.
- La ubicación donde se asignará la memoria para variables en tiempo de ejecución.
- Los tipos de operaciones permitidas.

El compilador usa información de tipo para garantizar que todas las operaciones que se realizan en el código cuentan *con seguridad de tipos*. Por ejemplo, si declara una variable de tipo int, el compilador le permite usar la variable en operaciones de suma y resta. Si intenta realizar esas mismas operaciones en una variable de tipo bool, el compilador genera un error, como se muestra en el siguiente ejemplo:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
>
> Los desarrolladores de C y C++ deben tener en cuenta que, en C#, bool no se puede convertir en int.

El compilador inserta la información de tipo en el archivo ejecutable como metadatos. Common Language Runtime (CLR) usa esos metadatos en tiempo de ejecución para garantizar aún más la seguridad de tipos cuando asigna y reclama memoria.

Especificar tipos en declaraciones de variable

Cuando declare una variable o constante en un programa, debe especificar su tipo o usar la palabra clave var para que el compilador infiera el tipo. En el ejemplo siguiente se muestran algunas declaraciones de variable que utilizan tanto tipos numéricos integrados como tipos complejos definidos por el usuario:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Los tipos de parámetros de método y valores devueltos se especifican en la firma del método. En la siguiente firma se muestra un método que requiere una variable int como argumento de entrada y devuelve una cadena:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

Tras declarar una variable, no se puede volver a declarar con un nuevo tipo y no se le puede asignar un valor que no sea compatible con su tipo declarado. Por ejemplo, no puede declarar un valor int y, luego, asignarle un valor booleano de `true`. En cambio, los valores se pueden convertir en otros tipos, por ejemplo, cuando se asignan a variables nuevas o se pasan como argumentos de método. El compilador realiza automáticamente una *conversión de tipo* que no da lugar a una pérdida de datos. Una conversión que pueda dar lugar a la pérdida de datos requiere un valor *cast* en el código fuente.

Para obtener más información, consulte Conversiones de tipos.

Tipos integrados

C# proporciona un conjunto estándar de tipos numéricos integrados para representar números enteros, valores de punto flotante, expresiones booleanas, caracteres de texto, valores decimales y otros tipos de datos. También hay tipos **string** y **object** integrados. Están disponibles para su uso en cualquier programa de C#. Para obtener una lista completa de los tipos integrados, vea Tipos integrados.

Tipos personalizados

Las construcciones struct, class, interface y enum se usan para crear sus propios tipos personalizados. La biblioteca de clases .NET Framework es en sí misma una colección de tipos personalizados proporcionados por Microsoft que puede usar en sus propias aplicaciones. De forma predeterminada, los tipos usados con más frecuencia en la biblioteca de clases están disponibles en cualquier programa de C#. Otros están disponibles solo cuando agrega explícitamente una referencia de proyecto al ensamblado en el que se definen. Una vez que el compilador tenga una referencia al ensamblado, puede declarar variables (y constantes) de los tipos declarados en dicho ensamblado en el código fuente.

Tipos genéricos

Los tipos se pueden declarar con uno o varios *parámetros de tipo* que actúan como un marcador de posición para el tipo real (el *tipo concreto*) que proporcionará el código de cliente cuando cree una instancia del tipo. Estos tipos se denominan *tipos genéricos*. Por ejemplo, el tipo de .NET Framework List<T> tiene un parámetro de tipo al que,

por convención, se le denomina *T*. Cuando crea una instancia del tipo, especifica el tipo de los objetos que contendrá la lista, por ejemplo, la cadena:

```
List<string> strings = new List<string>();
```

El uso del parámetro de tipo permite reutilizar la misma clase para incluir cualquier tipo de elemento, sin necesidad de convertir cada elemento en object. Las clases de colección genéricas se denominan *colecciones fuertemente tipadas* porque el compilador conoce el tipo específico de los elementos de la colección y puede generar un error en tiempo de compilación si, por ejemplo, intenta agregar un valor entero al objeto `strings` del ejemplo anterior. Para más información, vea Genéricos.

Tipos implícitos, tipos anónimos y tipos de tupla

Como se ha mencionado anteriormente, puede asignar implícitamente un tipo a una variable local (pero no miembros de clase) mediante la palabra clave var. La variable sigue recibiendo un tipo en tiempo de compilación, pero este lo proporciona el compilador. Para obtener más información, consulte Variables locales con asignación implícita de tipos.

En algunos casos, resulta conveniente crear un tipo con nombre para conjuntos sencillos de valores relacionados que no quiera almacenar ni pasar fuera de los límites del método. Puede crear *tipos anónimos* para este fin. Para obtener más información, consulte Tipos anónimos (Guía de programación de C#).

Es habitual querer devolver más de un valor de un método. Puede crear *tipos de tupla* que devuelven varios valores en una única llamada de método. Para más información, consulte Tuplas.

Common Type System

Es importante entender dos aspectos fundamentales sobre el sistema de tipos en .NET Framework:

- Es compatible con el principio de herencia. Los tipos pueden derivarse de otros tipos, denominados *tipos base*. El tipo derivado hereda (con algunas restricciones), los métodos, las propiedades y otros miembros del tipo base. A su vez, el tipo base puede derivarse de algún otro tipo, en cuyo caso el tipo derivado hereda los miembros de ambos tipos base en su jerarquía de herencia. Todos los tipos, incluidos los tipos numéricos integrados como Int32 (palabra clave de C#: `int`), se derivan en última instancia de un único tipo base, que es Object (palabra clave de C#: `object`). Esta jerarquía de tipos unificada se denomina Common Type System (CTS). Para obtener más información sobre la herencia en C#, consulte Herencia.
- En CTS, cada tipo se define como un *tipo de valor* o un *tipo de referencia*. Esto incluye todos los tipos personalizados de la biblioteca de clases .NET y también sus propios tipos definidos por el usuario. Los tipos que se definen mediante la palabra clave `struct` o `enum` son tipos de valor. Para más información sobre los tipos de valor, vea Tipos de valor. Los tipos que se definen mediante el uso de la palabra clave class son tipos de referencia. Para obtener más información sobre los tipos de referencia, consulte Classes (Clases). Los tipos de referencia y los tipos de valor tienen distintas reglas de tiempo de compilación y distintos comportamientos de tiempo de ejecución.

Vea también

- Tipos de estructura
- Tipos de enumeración
- Clases

Clases (Guía de programación de C#)

29/06/2020 • 9 minutes to read • Edit Online

Tipos de referencia

Un tipo que se define como una clase, es un *tipo de referencia*. Al declarar una variable de un tipo de referencia en tiempo de ejecución, esta contendrá el valor null hasta que se cree expresamente una instancia de la clase mediante el operador new o se le asigne un objeto de un tipo compatible que se ha creado en otro lugar, tal y como se muestra en el ejemplo siguiente:

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.
MyClass mc2 = mc;
```

Cuando se crea el objeto, se asigna suficiente memoria en el montón administrado para ese objeto específico y la variable solo contiene una referencia a la ubicación de dicho objeto. Los tipos del montón administrado producen sobrecarga cuando se asignan y cuando los reclama la función de administración de memoria automática de CLR, conocida como *recolección de elementos no utilizados*. En cambio, la recolección de elementos no utilizados también está muy optimizada y no crea problemas de rendimiento en la mayoría de los escenarios. Para obtener más información sobre la recolección de elementos no utilizados, vea Administración automática de la memoria y recolección de elementos no utilizados.

Declarar clases

Las clases se declaran mediante la palabra clave class seguida por un identificador único, como se muestra en el siguiente ejemplo:

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

La palabra clave `class` va precedida del nivel de acceso. Como en este caso se usa public, cualquier usuario puede crear instancias de esta clase. El nombre de la clase sigue a la palabra clave `class`. El nombre de la clase debe ser un nombre de identificador de C# válido. El resto de la definición es el cuerpo de la clase, donde se definen los datos y el comportamiento. Los campos, las propiedades, los métodos y los eventos de una clase se denominan de manera colectiva *miembros de clase*.

Creación de objetos

Aunque a veces se usan indistintamente, una clase y un objeto son cosas diferentes. Una clase define un tipo de objeto, pero no es un objeto en sí. Un objeto es una entidad concreta basada en una clase y, a veces, se conoce como una instancia de una clase.

Los objetos se pueden crear usando la palabra clave new, seguida del nombre de la clase en la que se basará el objeto, como en este ejemplo:

```
Customer object1 = new Customer();
```

Cuando se crea una instancia de una clase, se vuelve a pasar al programador una referencia al objeto. En el ejemplo anterior, `object1` es una referencia a un objeto que se basa en `Customer`. Esta referencia apunta al objeto nuevo, pero no contiene los datos del objeto. De hecho, puede crear una referencia de objeto sin tener que crear ningún objeto:

```
Customer object2;
```

No se recomienda crear referencias de objeto como esta, que no hace referencia a ningún objeto, ya que, si se intenta obtener acceso a un objeto a través de este tipo de referencia, se producirá un error en tiempo de ejecución. Pero dicha referencia puede haberse creado para hacer referencia a un objeto, ya sea creando un nuevo objeto o asignándola a un objeto existente, como en el siguiente ejemplo:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

Este código crea dos referencias de objeto que hacen referencia al mismo objeto. Por lo tanto, los cambios efectuados en el objeto mediante `object3` se reflejan en los usos posteriores de `object4`. Dado que los objetos basados en clases se tratan por referencia, las clases se denominan "tipos de referencia".

Herencia de clases

Las clases admiten completamente la *herencia*, una característica fundamental de la programación orientada a objetos. Al crear una clase, puede heredar de cualquier otra interfaz o clase que no esté definida como sealed; y otras clases pueden heredar de su clase e invalidar los métodos virtuales de la clase.

La herencia se consigue mediante una *derivación*, en la que se declara una clase mediante una *clase base*, desde la que hereda los datos y el comportamiento. Una clase base se especifica anexando dos puntos y el nombre de la clase base seguido del nombre de la clase derivada, como en el siguiente ejemplo:

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

Cuando una clase declara una clase base, hereda todos los miembros de la clase base excepto los constructores. Para obtener más información, vea Herencia.

A diferencia de C++, una clase de C# solo puede heredar directamente de una clase base. En cambio, dado que una clase base puede heredar de otra clase, una clase podría heredar indirectamente varias clases base. Además, una clase puede implementar directamente más de una interfaz. Para obtener más información, vea Interfaces.

Una clase puede declararse abstracta. Una clase abstracta contiene métodos abstractos que tienen una definición de firma, pero no tienen ninguna implementación. No se pueden crear instancias de las clases abstractas. Solo se pueden usar a través de las clases derivadas que implementan los métodos abstractos. Por el contrario, la clase sealed no permite que otras clases se deriven de ella. Para más información, vea Clases y miembros de clase abstractos y sellados.

Las definiciones de clase se pueden dividir entre distintos archivos de código fuente. Para más información, vea Clases y métodos parciales.

Ejemplo

En el ejemplo siguiente se define una clase pública que contiene una propiedad implementada automáticamente, un método y un método especial denominado constructor. Para obtener más información, consulta los temas Propiedades, Métodos y Constructores. Luego, se crea una instancia de las instancias de la clase con la palabra clave `new`.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#

- Programación orientada a objetos
- Polimorfismo
- Nombres de identificador
- Miembros
- Métodos
- Constructores
- Finalizadores
- Objects

Tipos de tupla de C#

29/06/2020 • 37 minutes to read • Edit Online

Las tuplas de C# son tipos que se definen mediante una sintaxis ligera. Entre otras ventajas, incluyen una sintaxis más sencilla, reglas para conversiones en función de un número (denominadas cardinalidad) y tipos de elementos y reglas coherentes para copias, pruebas de igualdad y asignaciones. Como contrapartida, las tuplas no admiten algunas de las expresiones orientadas a objetos que se asocian a la herencia. Puede obtener información general en la sección sobre tuplas del artículo Novedades de C# 7.0.

En este artículo conocerá las reglas del lenguaje que rigen las tuplas en C# 7.0 y versiones posteriores, distintas formas de usarlas y una guía inicial sobre cómo trabajar con tuplas.

> **NOTE**
>
> Las nuevas características de tupla requieren los tipos ValueTuple. Debe agregar el paquete NuGet `System.ValueTuple` para usarlo en plataformas que no incluyen los tipos.
>
> Esto es similar a otras características del lenguaje que se basan en tipos que se han proporcionado en el marco. Algunos ejemplos incluyen `async` y `await` que se basan en la interfaz `INotifyCompletion`, y LINQ que se basa en `IEnumerable<T>`. En cambio, el mecanismo de entrega está cambiando a medida que .NET está pasando a ser más independiente de las plataformas. Puede que .NET Framework no proporcione siempre la misma cadencia que el compilador de lenguaje. Cuando las nuevas características del lenguaje se basan en tipos nuevos, esos tipos estarán disponibles como paquetes NuGet cuando se proporcionen las características del lenguaje. Como estos tipos nuevos se agregan a la API de .NET Standard y se proporcionan como parte del marco, el requisito del paquete NuGet se quitará.

Empecemos por las razones para agregar nueva compatibilidad de tupla. Los métodos devuelven un solo objeto. Las tuplas permiten empaquetar varios valores en ese único objeto más fácilmente.

.NET Framework ya incluye clases `Tuple` genéricas. Pero estas clases planteaban dos importantes limitaciones. En primer lugar, las clases `Tuple` denominan a sus propiedades `Item1`, `Item2`, etc. Esos nombres no incluyen ninguna información semántica. El uso de estos tipos `Tuple` no permite comunicar el significado de cada una de las propiedades. Las nuevas características de lenguaje permiten declarar y utilizar nombres semánticamente significativos para los elementos de una tupla.

Las clases `Tuple` provocan más problemas de rendimiento porque son tipos de referencia. El uso de uno de los tipos `Tuple` implica la asignación de objetos. En rutas de acceso activas, la asignación de muchos objetos pequeños puede suponer un importante impacto en el rendimiento de la aplicación. Por lo tanto, la compatibilidad del lenguaje para tuplas aprovecha los nuevos struct `ValueTuple`.

Para evitar esas deficiencias, podría crear un `class` o `struct` que incluya varios elementos. Lamentablemente, esto representa más trabajo para el usuario y oculta la intención del diseño. La creación de un `struct` o `class` implica que se define un tipo con datos y comportamiento. Muchas veces, simplemente quiere almacenar varios valores en un solo objeto.

Las características del lenguaje y los structs genéricos `ValueTuple` aplican la regla que establece que no se puede agregar ningún comportamiento (métodos) a estos tipos de tupla. Todos los tipos `ValueTuple` son *structs mutables*. Cada campo de miembro es un campo público. Eso los hace muy ligeros. Pero indica que no se deben usar tuplas cuando la inmutabilidad es importante.

Las tuplas son contenedores de datos más sencillos y flexibles que los tipos `class` y `struct`. Examinemos esas diferencias.

Tuplas con nombre y sin nombre

El struct `ValueTuple` incluye campos denominados `Item1`, `Item2`, `Item3`, etc., similares a las propiedades definidas en los tipos `Tuple` existentes. Estos nombres son los únicos que se pueden usar en *tuplas sin nombre*. Si no proporciona ningún nombre de campo alternativo para una tupla, ha creado una tupla sin nombre:

```
var unnamed = ("one", "two");
```

La tupla del ejemplo anterior se ha inicializado con constantes literales y no tienen nombres de elementos creados mediante *proyecciones de nombre de campo de tupla* en C# 7.1.

Pero al inicializar una tupla, se pueden usar nuevas características del lenguaje que asignen nombres mejores a cada campo. Así se crea una *tupla con nombre*. Las tuplas con nombre todavía tienen elementos denominados `Item1`, `Item2`, `Item3`, etc. Pero también tienen sinónimos para cualquiera de esos elementos a los que haya asignado un nombre. Cree una tupla con nombre especificando los nombres de cada elemento. Una forma consiste en especificar los nombres como parte de la inicialización de la tupla:

```
var named = (first: "one", second: "two");
```

Estos sinónimos los controlan el compilador y el lenguaje para que se puedan usar tuplas con nombre de manera eficaz. Los IDE y los editores pueden leer estos nombres semánticos con las API de Roslyn. Puede hacer referencia a los elementos de una tupla con nombre por esos nombres semánticos en cualquier lugar de un mismo ensamblado. El compilador reemplaza los nombres que ha definido con `Item*` equivalentes al generar la salida compilada. El Lenguaje Intermedio de Microsoft (MSIL) compilado no incluye los nombres que se hayan asignado a estos elementos.

A partir de C# 7.1, se pueden proporcionar nombres de campo a una tupla desde las variables que se utilizan para inicializar la tupla. Esto se conoce como **inicializadores de proyección de tupla** . El código siguiente crea una tupla denominada `accumulation` con elementos `count` (un entero) y `sum` (un doble).

```
var sum = 12.5;
var count = 5;
var accumulation = (count, sum);
```

El compilador debe comunicar esos nombres creados por el usuario para tuplas que se devuelvan de métodos o propiedades públicos. En esos casos, el compilador agrega un atributo TupleElementNamesAttribute en el método. Este atributo contiene una propiedad de lista TransformNames que contiene los nombres asignados a cada uno de los elementos de la tupla.

> **NOTE**
>
> Las herramientas de desarrollo, como Visual Studio, también leen esos metadatos y proporcionan IntelliSense y otras características con los nombres de campo de metadatos.

Es importante comprender estos aspectos fundamentales subyacentes de las nuevas tuplas y el tipo `ValueTuple` para entender las reglas de asignación de tuplas con nombre entre sí.

Inicializadores de proyección de tupla

En general, los inicializadores de proyección de tupla funcionan con los nombres de variable o de campo desde el lado derecho de una instrucción de inicialización de tupla. Si no se proporciona un nombre explícito,

tiene prioridad sobre cualquier nombre proyectado. Por ejemplo, en el inicializador siguiente, los elementos son `explicitFieldOne` y `explicitFieldTwo`, no `localVariableOne` y `localVariableTwo`:

```
var localVariableOne = 5;
var localVariableTwo = "some text";

var tuple = (explicitFieldOne: localVariableOne, explicitFieldTwo: localVariableTwo);
```

Para cualquier campo en el que no se proporcione un nombre explícito, se proyecta un nombre implícito aplicable. No hay ningún requisito para proporcionar nombres semánticos, ya sea explícita o implícitamente. El inicializador siguiente tiene nombres de campo `Item1`, cuyo valor es `42` y `stringContent`, cuyo valor es "The answer to everything":

```
var stringContent = "The answer to everything";
var mixedTuple = (42, stringContent);
```

Hay dos condiciones donde los nombres de campo de candidato no se proyectan en el campo de la tupla:

1. Cuando el nombre del candidato es un nombre de tupla reservado. Los ejemplos incluyen `Item3`, `ToString` o `Rest`.
2. Cuando el nombre del candidato es un duplicado de otro nombre de campo de tupla, ya sea explícita o implícita.

Estas condiciones evitan la ambigüedad. Estos nombres podrían causar una ambigüedad si se usan como nombres de campo para un campo de una tupla. Ninguna de estas condiciones causan errores en tiempo de compilación. En su lugar, los elementos sin nombres proyectados no tienen nombres semánticos proyectados para ellos. Los ejemplos siguientes explican estas condiciones:

```
var ToString = "This is some text";
var one = 1;
var Item1 = 5;
var projections = (ToString, one, Item1);
// Accessing the first field:
Console.WriteLine(projections.Item1);
// There is no semantic name 'ToString'
// Accessing the second field:
Console.WriteLine(projections.one);
Console.WriteLine(projections.Item2);
// Accessing the third field:
Console.WriteLine(projections.Item3);
// There is no semantic name 'Item1`.

var pt1 = (X: 3, Y: 0);
var pt2 = (X: 3, Y: 4);

var xCoords = (pt1.X, pt2.X);
// There are no semantic names for the fields
// of xCoords.

// Accessing the first field:
Console.WriteLine(xCoords.Item1);
// Accessing the second field:
Console.WriteLine(xCoords.Item2);
```

Estas situaciones no causan errores de compilador dado que sería un cambio importante para el código escrito con C# 7.0, cuando las proyecciones del nombre de campo de tupla no estaban disponibles.

Igualdad y tuplas

Comenzando con C# 7.3, los tipos de tupla admiten los operadores `==` y `!=`. Estos operadores funcionan comparando cada uno de los miembros del argumento izquierdo con los miembros del argumento derecho en orden. Estas comparaciones cortocircuitan. Dejarán de evaluar a los miembros en cuanto un par no sea igual. Los siguientes ejemplos de código usan `==`, pero todas las reglas de comparación se aplican a `!=`. En el siguiente ejemplo de código se muestra una comparación de igualdad para dos pares de enteros:

```
var left = (a: 5, b: 10);
var right = (a: 5, b: 10);
Console.WriteLine(left == right); // displays 'true'
```

Hay varias reglas que hacen que las pruebas de igualdad de tupla sean más prácticas. La igualdad de tupla realiza conversiones elevadas si una de las tuplas es una tupla que admite valores NULL, como se muestra en el siguiente código:

```
var left = (a: 5, b: 10);
var right = (a: 5, b: 10);
(int a, int b)? nullableTuple = right;
Console.WriteLine(left == nullableTuple); // Also true
```

La igualdad de tupla también realiza conversiones implícitas en cada uno de los miembros de ambas tuplas. Entre estas se incluyen conversiones elevadas, conversiones de ampliación u otras conversiones implícitas. En los siguientes ejemplos se muestra que una tupla de 2 entera se puede comparar con una tupla de 2 larga debido a la conversión implícita de entero a largo:

```
// lifted conversions
var left = (a: 5, b: 10);
(int? a, int? b) nullableMembers = (5, 10);
Console.WriteLine(left == nullableMembers); // Also true

// converted type of left is (long, long)
(long a, long b) longTuple = (5, 10);
Console.WriteLine(left == longTuple); // Also true

// comparisons performed on (long, long) tuples
(long a, int b) longFirst = (5, 10);
(int a, long b) longSecond = (5, 10);
Console.WriteLine(longFirst == longSecond); // Also true
```

Los nombres de los miembros de las tuplas no participan en las pruebas para la igualdad. Sin embargo, si uno de los operandos es un literal de tupla con nombres explícitos, el compilador genera la advertencia CS8383 si esos nombres no coinciden con los nombres del otro operando. Cuando ambos operandos son literales de tupla, la advertencia está en el operando derecho como se muestra en el siguiente ejemplo:

```
(int a, string b) pair = (1, "Hello");
(int z, string y) another = (1, "Hello");
Console.WriteLine(pair == another); // true. Member names don't participate.
Console.WriteLine(pair == (z: 1, y: "Hello")); // warning: literal contains different member names
```

Por último, las tuplas pueden contener tuplas anidadas. La igualdad de tupla compara la "forma" de cada operando a través de tuplas anidadas como se muestra en el siguiente ejemplo:

```
(int, (int, int)) nestedTuple = (1, (2, 3));
Console.WriteLine(nestedTuple == (1, (2, 3)) );
```

Es un error de tiempo de compilación comparar la igualdad (o desigualdad) de dos tuplas cuando tienen

formas diferentes. El compilador no intenta ninguna deconstrucción de tuplas anidadas para compararlas.

Asignación y tuplas

El lenguaje admite la asignación entre tipos de tupla que tienen el mismo número de elementos, donde cada elemento del lado derecho se puede convertir de forma implícita en su elemento del lado izquierdo correspondiente. Otras conversiones no se tienen en cuenta para las asignaciones. Es un error de tiempo de compilación asignar una tupla a otra cuando tienen formas diferentes. El compilador no intentará ninguna deconstrucción de tuplas anidadas para asignarlas. Echemos un vistazo a los tipos de asignaciones que se permiten entre los tipos de tupla.

Tenga en cuenta estas variables que se usan en los ejemplos siguientes:

```
// The 'arity' and 'shape' of all these tuples are compatible.
// The only difference is the field names being used.
var unnamed = (42, "The meaning of life");
var anonymous = (16, "a perfect square");
var named = (Answer: 42, Message: "The meaning of life");
var differentNamed = (SecretConstant: 42, Label: "The meaning of life");
```

Las dos primeras variables, `unnamed` y `anonymous`, no tienen nombres semánticos proporcionados para los elementos. Los nombres de campo son `Item1` y `Item2`. Las dos últimas variables, `named` y `differentName`, tienen nombres semánticos asignados a los elementos. Estas dos tuplas tienen nombres diferentes para los elementos.

Estas cuatro tuplas tienen el mismo número de elementos (denominados "cardinalidad") y los tipos de esos elementos son idénticos. Por consiguiente, todas estas asignaciones funcionan:

```
unnamed = named;

named = unnamed;
// 'named' still has fields that can be referred to
// as 'answer', and 'message':
Console.WriteLine($"{named.Answer}, {named.Message}");

// unnamed to unnamed:
anonymous = unnamed;

// named tuples.
named = differentNamed;
// The field names are not assigned. 'named' still has
// fields that can be referred to as 'answer' and 'message':
Console.WriteLine($"{named.Answer}, {named.Message}");

// With implicit conversions:
// int can be implicitly converted to long
(long, string) conversion = named;
```

Observe que los nombres de las tuplas no se asignan. Los valores de los elementos se asignan según el orden de los campos de la tupla.

Las tuplas de diferentes tipos o números de elementos no son asignables:

```
// Does not compile.
// CS0029: Cannot assign Tuple(int,int,int) to Tuple(int, string)
var differentShape = (1, 2, 3);
named = differentShape;
```

Tuplas como valores devueltos del método

Uno de los usos más comunes de tuplas es como un valor devuelto del método. Examinemos un ejemplo. Tenga en cuenta este método que calcula la desviación estándar para una secuencia de números:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    // Step 1: Compute the Mean:
    var mean = sequence.Average();

    // Step 2: Compute the square of the differences between each number
    // and the mean:
    var squaredMeanDifferences = from n in sequence
                                 select (n - mean) * (n - mean);
    // Step 3: Find the mean of those squared differences:
    var meanOfSquaredDifferences = squaredMeanDifferences.Average();

    // Step 4: Standard Deviation is the square root of that mean:
    var standardDeviation = Math.Sqrt(meanOfSquaredDifferences);
    return standardDeviation;
}
```

> **NOTE**
>
> En estos ejemplos se calcula la desviación estándar de muestra sin corregir. La fórmula de desviación estándar de muestra corregida dividiría la suma de las diferencias al cuadrado de la media por (N-1) en lugar de N, como hace el método de extensión `Average`. Para obtener más detalles sobre las diferencias entre estas fórmulas de desviación estándar, consulte un texto de estadísticas.

El código anterior sigue la fórmula clásica para la desviación estándar. Genera la respuesta correcta, pero es una implementación poco eficaz. Este método enumera la secuencia dos veces: una para generar el promedio y otra para generar el promedio del cuadrado de la diferencia del promedio. (Recuerde que las consultas LINQ se evalúan de forma diferida, por lo que el cálculo de las diferencias de la media y el promedio de estas diferencias crea una sola enumeración).

Hay una fórmula alternativa que calcula la desviación estándar mediante una sola enumeración de la secuencia. Este cálculo genera dos valores cuando enumera la secuencia: la suma de todos los elementos de la secuencia y la suma del valor de cada cuadrado:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    double sum = 0;
    double sumOfSquares = 0;
    double count = 0;

    foreach (var item in sequence)
    {
        count++;
        sum += item;
        sumOfSquares += item * item;
    }

    var variance = sumOfSquares - sum * sum / count;
    return Math.Sqrt(variance / count);
}
```

Esta versión enumera la secuencia exactamente una vez. Pero no se trata de código reutilizable. A medida que siga trabajando, verá que muchos cálculos estadísticos diferentes usan el número de elementos de la secuencia, la suma de la secuencia y la suma de los cuadrados de la secuencia. Vamos a refactorizar este

método y escribir un método de utilidad que genera esos tres valores. Los tres valores pueden devolverse como tupla.

Vamos a actualizar este método para que los tres valores calculados durante la enumeración se almacenen en una tupla. Se crea esta versión:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    var computation = (Count: 0, Sum: 0.0, SumOfSquares: 0.0);

    foreach (var item in sequence)
    {
        computation.Count++;
        computation.Sum += item;
        computation.SumOfSquares += item * item;
    }

    var variance = computation.SumOfSquares - computation.Sum * computation.Sum / computation.Count;
    return Math.Sqrt(variance / computation.Count);
}
```

La compatibilidad de refactorización de Visual Studio facilita la extracción de la funcionalidad de las estadísticas principales en un método privado. Le ofrece un método `private static` que devuelve el tipo de tupla con los tres valores de `Sum`, `SumOfSquares` y `Count`:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    (int Count, double Sum, double SumOfSquares) computation = ComputeSumAndSumOfSquares(sequence);

    var variance = computation.SumOfSquares - computation.Sum * computation.Sum / computation.Count;
    return Math.Sqrt(variance / computation.Count);
}

private static (int Count, double Sum, double SumOfSquares) ComputeSumAndSumOfSquares(IEnumerable<double>
sequence)
{
    var computation = (count: 0, sum: 0.0, sumOfSquares: 0.0);

    foreach (var item in sequence)
    {
        computation.count++;
        computation.sum += item;
        computation.sumOfSquares += item * item;
    }

    return computation;
}
```

El lenguaje permite un par de opciones más que puede usar si quiere realizar algunas modificaciones rápidas manualmente. En primer lugar, puede usar la declaración `var` para inicializar el resultado de la tupla de la llamada al método `ComputeSumAndSumOfSquares`. También puede crear tres variables discretas dentro del método `ComputeSumAndSumOfSquares`. La versión final se muestra en el siguiente código:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    var computation = ComputeSumAndSumOfSquares(sequence);

    var variance = computation.SumOfSquares - computation.Sum * computation.Sum / computation.Count;
    return Math.Sqrt(variance / computation.Count);
}

private static (int Count, double Sum, double SumOfSquares) ComputeSumAndSumOfSquares(IEnumerable<double>
sequence)
{
    double sum = 0;
    double sumOfSquares = 0;
    int count = 0;

    foreach (var item in sequence)
    {
        count++;
        sum += item;
        sumOfSquares += item * item;
    }

    return (count, sum, sumOfSquares);
}
```

Esta versión final puede usarse en cualquier método que necesite esas tres variables o cualquier subconjunto de ellas.

El lenguaje admite otras opciones de administración de los nombres de los elementos en estos métodos de devolución de tuplas.

Puede quitar los nombres de campo de la declaración de valor devuelto y devolver una tupla sin nombre:

```
private static (double, double, int) ComputeSumAndSumOfSquares(IEnumerable<double> sequence)
{
    double sum = 0;
    double sumOfSquares = 0;
    int count = 0;

    foreach (var item in sequence)
    {
        count++;
        sum += item;
        sumOfSquares += item * item;
    }

    return (sum, sumOfSquares, count);
}
```

Los campos de esta tupla se denominan `Item1`, `Item2` y `Item3`. Se recomienda proporcionar nombres semánticos a los elementos de tuplas devueltas por los métodos.

Otra expresión donde las tuplas pueden ser útiles es cuando crea consultas LINQ. El resultado proyectado final suele contener algunas de las propiedades de los objetos que se seleccionan, pero no todas.

Tradicionalmente, los resultados de la consulta se proyectarían en una secuencia de objetos que fueran un tipo anónimo. Eso suponía muchas limitaciones, sobre todo porque a los tipos anónimos no se les podía asignar nombre cómodamente en el tipo de valor devuelto para un método. Las alternativas que usaban `object` o `dynamic` como tipo de resultado acarreaban importantes costos de rendimiento.

Devolver una secuencia de tipo de tupla es fácil, y los nombres y tipos de los elementos están disponibles en tiempo de compilación y a través de herramientas de IDE. Por ejemplo, consideremos una aplicación de

tareas pendientes. Puede definir una clase similar a la siguiente para representar una sola entrada de la lista de tareas pendientes:

```
public class ToDoItem
{
    public int ID { get; set; }
    public bool IsDone { get; set; }
    public DateTime DueDate { get; set; }
    public string Title { get; set; }
    public string Notes { get; set; }
}
```

Puede que las aplicaciones móviles admitan una forma compacta de las tareas pendientes actuales que solo muestra el título. Esa consulta LINQ haría una proyección que solo incluya el identificador y el título. Un método que devuelve una secuencia de tuplas expresa bien ese diseño:

```
internal IEnumerable<(int ID, string Title)> GetCurrentItemsMobileList()
{
    return from item in AllItems
           where !item.IsDone
           orderby item.DueDate
           select (item.ID, item.Title);
}
```

NOTE

En C# 7.1, las proyecciones de tupla permiten crear tuplas con nombre mediante elementos, de forma similar a la propiedad en tipos anónimos. En el código anterior, la instrucción `select` de la proyección de consultas crea una tupla con los elementos `ID` y `Title`.

La tupla con nombre puede ser parte de la firma. Permite que el compilador y las herramientas de IDE usen comprobación de tipo estáticos para ver que el resultado se usa correctamente. La tupla con nombre también incluye información de tipos estáticos, por lo que no hay que usar costosas características en tiempo de ejecución tales como la reflexión o los enlaces dinámicos para trabajar con los resultados.

Deconstrucción

Puede desempaquetar todos los elementos de una tupla *deconstruyendo* la tupla devuelta por un método. Existen tres enfoques diferentes para deconstruir tuplas. En primer lugar, se puede declarar explícitamente el tipo de cada campo entre paréntesis para crear variables discretas para cada uno de los elementos de la tupla:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    (int count, double sum, double sumOfSquares) = ComputeSumAndSumOfSquares(sequence);

    var variance = sumOfSquares - sum * sum / count;
    return Math.Sqrt(variance / count);
}
```

También puede declarar variables con tipo implícito para cada campo de una tupla mediante la palabra clave `var` fuera de los paréntesis:

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    var (sum, sumOfSquares, count) = ComputeSumAndSumOfSquares(sequence);

    var variance = sumOfSquares - sum * sum / count;
    return Math.Sqrt(variance / count);
}
```

También es válido usar la palabra clave `var` con alguna de las declaraciones de variable, o todas, dentro de los paréntesis.

```
(double sum, var sumOfSquares, var count) = ComputeSumAndSumOfSquares(sequence);
```

No se puede usar un tipo específico fuera de los paréntesis, aunque todos los campos de la tupla tengan el mismo tipo.

También puede deconstruir tuplas con declaraciones existentes:

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

WARNING

No se pueden mezclar declaraciones existentes con declaraciones incluidas en paréntesis. Por ejemplo, no se permite lo siguiente: `(var x, y) = MyMethod();`. Esto produce el error CS8184 porque *x* se declara dentro de los paréntesis e *y* se ha declarado previamente en otro lugar.

Deconstruir tipos definidos por el usuario

Cualquier tipo de tupla puede deconstruirse, tal y como se muestra anteriormente. También resulta fácil habilitar la deconstrucción en cualquier tipo definido por el usuario (clases, structs o incluso interfaces).

El autor del tipo puede definir uno o varios métodos `Deconstruct` que asignen valores a cualquier número de variables `out` que representen los elementos de datos que componen el tipo. Por ejemplo, el tipo `Person` siguiente define un método `Deconstruct` que deconstruye un objeto person en los elementos que representan el nombre y apellido:

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}
```

El método deconstruct permite la asignación desde un objeto `Person` en dos cadenas que representan las propiedades `FirstName` y `LastName`:

```
var p = new Person("Althea", "Goodwin");
var (first, last) = p;
```

Puede habilitar la deconstrucción incluso para tipos que no ha creado. El método `Deconstruct` puede ser un método de extensión que desempaqueta los miembros de datos accesibles de un objeto. En el ejemplo siguiente se muestra un tipo `Student`, derivado del tipo `Person` y un método de extensión que deconstruye un `Student` en tres variables, que representan el `FirstName`, el `LastName` y `GPA`:

```
public class Student : Person
{
    public double GPA { get; }
    public Student(string first, string last, double gpa) :
        base(first, last)
    {
        GPA = gpa;
    }
}

public static class Extensions
{
    public static void Deconstruct(this Student s, out string first, out string last, out double gpa)
    {
        first = s.FirstName;
        last = s.LastName;
        gpa = s.GPA;
    }
}
```

Un objeto `Student` ahora tiene dos métodos `Deconstruct` accesibles: el método de extensión declarado para tipos `Student` y el miembro del tipo `Person`. Ambos están en el ámbito, lo que permite que un `Student` se deconstruya en dos variables o tres. Si asigna a un alumno tres variables, se devuelven todas, el nombre, el apellido y el GPA. Si asigna a un alumno dos variables, solo se devuelven el nombre y el apellido.

```
var s1 = new Student("Cary", "Totten", 4.5);
var (fName, lName, gpa) = s1;
```

Se debe tener cuidado al definir varios métodos `Deconstruct` en una clase o una jerarquía de clases. Varios

métodos `Deconstruct` con el mismo número de parámetros `out` pueden generar ambigüedades rápidamente. Puede que los autores de llamadas no puedan llamar fácilmente al método `Deconstruct` que quieran.

En este ejemplo, la posibilidad de una llamada ambigua es mínima porque el método `Deconstruct` para `Person` tiene dos parámetros de salida y el método `Deconstruct` para `Student` tiene tres.

Los operadores de deconstrucción no participan en la igualdad de las pruebas. El ejemplo siguiente genera el error del compilador CS0019:

```
Person p = new Person("Althea", "Goodwin");
if (("Althea", "Goodwin") == p)
    Console.WriteLine(p);
```

El método `Deconstruct` podría convertir el objeto `Person``p` en una tupla que contiene dos cadenas, pero no se puede aplicar en el contexto de las pruebas de igualdad.

Tuplas como parámetros de salida

Las tuplas se pueden usar como parámetros `out` *en sí*. No se deben confundir con ninguna ambigüedad mencionada anteriormente en la sección Deconstrucción. En una llamada al método, solo necesita describir la forma de la tupla:

```
Dictionary<int, (int, string)> dict = new Dictionary<int, (int, string)>();
dict.Add(1, (234, "First!"));
dict.Add(2, (345, "Second"));
dict.Add(3, (456, "Last"));

// TryGetValue already demonstrates using out parameters
dict.TryGetValue(2, out (int num, string place) pair);

Console.WriteLine($"{pair.num}: {pair.place}");

/*
 * Output:
 * 345: Second
 */
```

Como alternativa, puede usar una tupla *sin nombre* y hacer referencia a sus campos como `Item1` y `Item2`:

```
dict.TryGetValue(2, out (int, string) pair);
// ...
Console.WriteLine($"{pair.Item1}: {pair.Item2}");
```

Conclusión

La compatibilidad con tuplas con nombre del nuevo lenguaje y la biblioteca hace que resulte mucho más fácil trabajar con diseños que usan estructuras de datos que almacenan varios elementos, pero no definen el comportamiento, como clases y structs. El uso de tuplas para esos tipos resulta fácil y conciso. Se tienen todas las ventajas de la comprobación de tipos estáticos, sin necesidad de crear tipos con la sintaxis más detallada de `class` o `struct`. Aun así, resultan muy útiles para métodos de utilidad que sean `private` o `internal`. Cree tipos definidos por el usuario, tipos `class` o `struct`, cuando los métodos públicos devuelvan un valor con varios elementos.

Deconstruir tuplas y otros tipos

20/04/2020 • 17 minutes to read • Edit Online

Una tupla proporciona una manera ligera de recuperar varios valores de una llamada de método. Pero una vez que recupere la tupla, deberá controlar sus elementos individuales. Como se muestra en el ejemplo siguiente, es complicado hacerlo elemento a elemento. El método `QueryCityData` devuelve una tupla de 3, y cada uno de sus elementos se asigna a una variable en una operación independiente.

```
using System;

public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

         // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

También puede ser complicado recuperar varios valores de campo y propiedad de un objeto, ya que tendrá que asignar un valor de campo o propiedad a una variable miembro a miembro.

A partir de C# 7.0, puede recuperar varios elementos de una tupla o recuperar varios valores de campo, de propiedad y calculados de un objeto en una sola operación *deconstruct*. Cuando se deconstruye una tupla, sus elementos se asignan a variables individuales. Cuando se deconstruye un objeto, los valores seleccionados se asignan a variables individuales.

Deconstruir una tupla

C# incluye compatibilidad integrada para deconstruir tuplas, lo que permite desempaquetar todos los elementos de una tupla en una sola operación. La sintaxis general para deconstruir una tupla es parecida a la sintaxis para definirla, ya que las variables a las que se va a asignar cada elemento se escriben entre paréntesis en el lado izquierdo de una instrucción de asignación. Por ejemplo, la instrucción siguiente asigna los elementos de una tupla de 4 a cuatro variables independientes:

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

Hay tres formas de deconstruir una tupla:

- Se puede declarar explícitamente el tipo de cada campo entre paréntesis. En el ejemplo siguiente se usa este enfoque para deconstruir la tupla de 3 devuelta por el método `QueryCityData`.

```
public static void Main()
{
    (string city, int population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- Puede usar la palabra clave `var` para que C# deduzca el tipo de cada variable. Debe colocar la palabra clave `var` fuera de los paréntesis. En el ejemplo siguiente se usa la inferencia de tipos al deconstruir la tupla de 3 devuelta por el método `QueryCityData`.

  ```
  public static void Main()
  {
      var (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  También se puede usar la palabra clave `var` individualmente con alguna de las declaraciones de variable, o todas, dentro de los paréntesis.

  ```
  public static void Main()
  {
      (string city, var population, var area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  Esto es complicado y no se recomienda.

- Por último, puede deconstruir la tupla en variables que ya se hayan declarado.

  ```
  public static void Main()
  {
      string city = "Raleigh";
      int population = 458880;
      double area = 144.8;

      (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

Tenga en cuenta que no se puede especificar un tipo determinado fuera de los paréntesis, aunque todos los campos de la tupla tengan el mismo tipo. Esto genera el error del compilador CS8136: “El formato de desconstrucción ‘var (...)’ no permite especificar un tipo determinado para ‘var’”.

Tenga en cuenta que también debe asignar cada elemento de la tupla a una variable. Si se omite algún elemento, el compilador genera el error CS8132: "No se puede deconstruir una tupla de 'x' elementos en 'y' variables".

Tenga en cuenta que no se pueden mezclar declaraciones y asignaciones en variables existentes en el lado izquierdo de una deconstrucción. El compilador genera el error CS8184: "Una deconstrucción no puede mezclar declaraciones y expresiones en el lado izquierdo", cuando los miembros incluyen variables existentes recién declaradas.

Deconstruir elementos de tupla con descartes

A menudo, cuando se deconstruye una tupla, solo interesan los valores de algunos elementos. A partir de C# 7.0, puede aprovechar la compatibilidad de C# con los *descartes*, que son variables de solo escritura cuyos valores se decide omitir. Los descartes suelen designarse mediante un carácter de guion bajo ("_") en una asignación. Puede descartar tantos valores como quiera; todos se representan mediante el descarte único `_`.

En el ejemplo siguiente se muestra el uso de tuplas con descartes. El método `QueryCityDataForYears` devuelve una tupla de 6 con el nombre de una ciudad, su superficie, un año, la población de la ciudad en ese año, un segundo año y la población de la ciudad en ese segundo año. En el ejemplo se muestra la evolución de la población entre esos dos años. De los datos disponibles en la tupla, no nos interesa la superficie de la ciudad, y conocemos el nombre de la ciudad y las dos fechas en tiempo de diseño. Como resultado, solo nos interesan los dos valores de población almacenados en la tupla, y podemos controlar los valores restantes como descartes.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Deconstruir tipos definidos por el usuario

C# no ofrece compatibilidad integrada para deconstruir tipos que no son de tupla. A pesar de ello, como autor de una clase, una estructura o una interfaz, puede permitir que las instancias del tipo se deconstruyan mediante la implementación de uno o varios métodos `Deconstruct`. El método no devuelve ningún valor, y cada valor que se va a deconstruir se indica mediante un parámetro out en la firma del método. Por ejemplo, el siguiente método `Deconstruct` de una clase `Person` devuelve el nombre de pila, el segundo nombre y los apellidos:

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

Después, puede deconstruir una instancia de la clase `Person` denominada `p` con una asignación similar a la siguiente:

```
var (fName, mName, lName) = p;
```

En el ejemplo siguiente se sobrecarga el método `Deconstruct` para devolver varias combinaciones de las propiedades de un objeto `Person`. Las sobrecargas individuales devuelven lo siguiente:

- El nombre de pila y los apellidos.
- El nombre de pila, los apellidos y el segundo nombre.
- El nombre de pila, los apellidos, el nombre de la ciudad y el nombre del estado.

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class Example
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Dado que se puede sobrecargar el método `Deconstruct` para reflejar los grupos de datos que se suelen extraer de un objeto, debe tener cuidado y definir métodos `Deconstruct` con firmas que sean distintivas y no resulten ambiguas. Puede resultar confuso usar varios métodos `Deconstruct` que tengan el mismo número de parámetros `out` o el mismo número y tipo de parámetros `out` en un orden diferente.

El método `Deconstruct` sobrecargado del ejemplo siguiente muestra un posible motivo de confusión. La primera sobrecarga devuelve el nombre de pila, el segundo nombre, los apellidos y la edad de un objeto `Person`, en ese

orden. La segunda sobrecarga devuelve información sobre el nombre acompañada únicamente de los ingresos anuales, pero el nombre de pila, el segundo nombre y los apellidos están en un orden diferente. Esto hace que se pueda confundir fácilmente el orden de los argumentos cuando se deconstruye una instancia `Person`.

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public DateTime DateOfBirth { get; set; }
    public Decimal AnnualIncome { get; set; }

    public void Deconstruct(out string fname, out string mname, out string lname, out int age)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;

        // calculate the person's age
        var today = DateTime.Today;
        age = today.Year - DateOfBirth.Year;
        if (DateOfBirth.Date > today.AddYears(-age))
            age--;
    }

    public void Deconstruct(out string lname, out string fname, out string mname, out decimal income)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
        income = AnnualIncome;
    }
}
```

Deconstruir un tipo definido por el usuario con descartes

Tal como haría con las tuplas, puede usar descartes para omitir los elementos seleccionados que haya devuelto un método `Deconstruct`. Cada descarte se define mediante una variable denominada "_". Una operación de deconstrucción única puede incluir varios descartes.

En el siguiente ejemplo se deconstruye un objeto `Person` en cuatro cadenas (el nombre propio, los apellidos, la ciudad y el estado), pero se descartan los apellidos y el estado.

```
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

Deconstruir un tipo definido por el usuario con un método de extensión

Aunque no haya creado una clase, una estructura o una interfaz por sí mismo, puede igualmente deconstruir objetos de ese tipo. Para ello, implemente uno o varios métodos de extensión `Deconstruct` que devuelvan los valores que le interesen.

En el ejemplo siguiente se definen dos métodos de extensión `Deconstruct` para la clase System.Reflection.PropertyInfo. El primero devuelve un conjunto de valores que indican las características de la propiedad, incluido su tipo, si es estática o de instancia, si es de solo lectura y si está indexada. El segundo indica la accesibilidad de la propiedad. Dado que la accesibilidad de los descriptores de acceso get y set puede diferir, los valores booleanos indican si la propiedad tiene descriptores de acceso get y set independientes y, si es así, si tienen la misma accesibilidad. Si solo hay un descriptor de acceso, o si los descriptores de acceso get y set tienen la misma accesibilidad, la variable `access` indica la accesibilidad de la propiedad en conjunto. En caso contrario, la accesibilidad de los descriptores de acceso get y set se indica mediante las variables `getAccess` y `setAccess`.

```
using System;
using System.Collections.Generic;
using System.Reflection;

public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                   out bool isReadOnly, out bool isIndexed,
                                   out Type propertyType)
    {
        var getter = p.GetMethod;

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool hasGetAndSet,
                                   out bool sameAccess, out string access,
                                   out string getAccess, out string setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
        if (p.CanRead)
            getter = p.GetMethod;

        MethodInfo setter = null;
        if (p.CanWrite)
            setter = p.SetMethod;

        if (setter != null && getter != null)
            hasGetAndSet = true;

        if (getter != null)
        {
            if (getter.IsPublic)
                getAccessTemp = "public";
            else if (getter.IsPrivate)
                getAccessTemp = "private";
            else if (getter.IsAssembly)
                getAccessTemp = "internal";
            else if (getter.IsFamily)
                getAccessTemp = "protected";
            else if (getter.IsFamilyOrAssembly)
                getAccessTemp = "protected internal";
```

```
        }

        if (setter != null)
        {
            if (setter.IsPublic)
                setAccessTemp = "public";
            else if (setter.IsPrivate)
                setAccessTemp = "private";
            else if (setter.IsAssembly)
                setAccessTemp = "internal";
            else if (setter.IsFamily)
                setAccessTemp = "protected";
            else if (setter.IsFamilyOrAssembly)
                setAccessTemp = "protected internal";
        }

        // Are the accessibility of the getter and setter the same?
        if (setAccessTemp == getAccessTemp)
        {
            sameAccess = true;
            access = getAccessTemp;
            getAccess = setAccess = String.Empty;
        }
        else
        {
            access = null;
            getAccess = getAccessTemp;
            setAccess = setAccessTemp;
        }
    }
}

public class Example
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name} property:");
        Console.WriteLine($"   PropertyType: {propType.Name}");
        Console.WriteLine($"   Static:       {isStatic}");
        Console.WriteLine($"   Read-only:    {isRO}");
        Console.WriteLine($"   Indexed:      {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance |
BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility, setAccessibility) = prop;
        Console.Write($"\nAccessibility of the {listType.FullName}.{prop.Name} property: ");

        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\n   The get accessor: {getAccessibility}");
            Console.WriteLine($"   The set accessor: {setAccessibility}");
        }
    }
}
// The example displays the following output:
//       The System.DateTime.Now property:
//          PropertyType: DateTime
//          Static:       True
//          Read-only:    True
//          Indexed:      False
```

```
//
//       Accessibility of the System.Collections.Generic.List`1.Item property: public
```

Vea también

- Descartes
- Tuplas

Interfaces (Guía de programación de C#)

09/04/2020 • 8 minutes to read • Edit Online

Una interfaz contiene las definiciones de un grupo de funcionalidades relacionadas que una clase o una estructura no abstracta deben implementar. Una interfaz puede definir métodos `static`, que deben tener una implementación. A partir de C# 8.0, una interfaz puede definir una implementación predeterminada de miembros. Una interfaz no puede declarar datos de instancia, como campos, propiedades implementadas automáticamente o eventos similares a las propiedades.

Mediante las interfaces puede incluir, por ejemplo, un comportamiento de varios orígenes en una clase. Esta capacidad es importante en C# porque el lenguaje no admite la herencia múltiple de clases. Además, debe usar una interfaz si desea simular la herencia de estructuras, porque no pueden heredar de otra estructura o clase.

Para definir una interfaz, deberá usar la palabra clave interface, tal y como se muestra en el ejemplo siguiente.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

El nombre de una interfaz debe ser un nombre de identificador de C# válido. Por convención, los nombres de interfaz comienzan con una letra `I` mayúscula.

Cualquier clase o estructura que implementa la interfaz IEquatable<T> debe contener una definición para un método Equals que coincida con la firma que la interfaz especifica. Como resultado, puede contar con una clase que implementa `IEquatable<T>` para contener un método `Equals` con el que una instancia de la clase puede determinar si es igual a otra instancia de la misma clase.

La definición de `IEquatable<T>` no facilita ninguna implementación para `Equals`. Una clase o estructura puede implementar varias interfaces, pero una clase solo puede heredar de una sola clase.

Para obtener más información sobre las clases abstractas, vea Clases y miembros de clase abstractos y sellados (Guía de programación de C#).

Las interfaces pueden contener propiedades, eventos, indizadores o métodos de instancia, o bien cualquier combinación de estos cuatro tipos de miembros. Las interfaces pueden contener constructores estáticos, campos, constantes u operadores. Para obtener vínculos a ejemplos, vea Secciones relacionadas. Una interfaz no puede contener campos de instancia, constructores de instancias ni finalizadores. Los miembros de la interfaz son públicos de forma predeterminada.

Para implementar un miembro de interfaz, el miembro correspondiente de la clase de implementación debe ser público, no estático y tener el mismo nombre y firma que el miembro de interfaz.

Cuando una clase o estructura implementa una interfaz, la clase o estructura debe proporcionar una implementación para todos los miembros que la interfaz declara sin proporcionar ninguna implementación predeterminada para ellos. Sin embargo, si una clase base implementa una interfaz, cualquier clase que se derive de la clase base hereda esta implementación.

En el siguiente ejemplo se muestra una implementación de la interfaz IEquatable<T>. La clase de implementación `Car` debe proporcionar una implementación del método Equals.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}
```

Las propiedades y los indizadores de una clase pueden definir descriptores de acceso adicionales para una propiedad o indizador que estén definidos en una interfaz. Por ejemplo, una interfaz puede declarar una propiedad que tenga un descriptor de acceso get. La clase que implementa la interfaz puede declarar la misma propiedad con un descriptor de acceso `get` y set. Sin embargo, si la propiedad o el indizador usan una implementación explícita, los descriptores de acceso deben coincidir. Para obtener más información sobre la implementación explícita, vea Implementación de interfaz explícita y Propiedades de interfaces.

Las interfaces pueden heredar de una o varias interfaces. La interfaz derivada hereda los miembros de sus interfaces base. Una clase que implementa una interfaz derivada debe implementar todos los miembros de esta, incluidos los de las interfaces base. Esa clase puede convertirse implícitamente en la interfaz derivada o en cualquiera de sus interfaces base. Una clase puede incluir una interfaz varias veces mediante las clases base que hereda o mediante las interfaces que otras interfaces heredan. Sin embargo, la clase puede proporcionar una implementación de una interfaz solo una vez y solo si la clase declara la interfaz como parte de la definición de la clase (`class ClassName : InterfaceName`). Si la interfaz se hereda porque se heredó una clase base que implementa la interfaz, la clase base proporciona la implementación de los miembros de la interfaz. Sin embargo, la clase derivada puede volver a implementar cualquier miembro de la interfaz virtual, en lugar de usar la implementación heredada. Cuando una interfaz declara una implementación predeterminada de un método, cualquier clase que implemente la interfaz en cuestión hereda esa implementación. Las implementaciones definidas en interfaces son virtuales y es posible que la clase que realice la implementación las invalide.

Una clase base también puede implementar miembros de interfaz mediante el uso de los miembros virtuales. En ese caso, una clase derivada puede cambiar el comportamiento de la interfaz reemplazando los miembros virtuales. Para obtener más información sobre los miembros virtuales, vea Polimorfismo.

Resumen de interfaces

Una interfaz tiene las propiedades siguientes:

- Normalmente, una interfaz es como una clase base abstracta con solo miembros abstractos. Cualquier clase o estructura que implementa la interfaz debe implementar todos sus miembros. Opcionalmente, una interfaz puede definir implementaciones predeterminadas para algunos o todos sus miembros.
- No se puede crear una instancia de una interfaz directamente. Sus miembros se implementan por medio de cualquier clase o estructura que implementa la interfaz.
- Una clase o estructura puede implementar varias interfaces. Una clase puede heredar una clase base y también implementar una o varias interfaces.

Secciones relacionadas

- Propiedades de interfaz
- Indizadores en Interfaces
- Procedimiento para implementar eventos de interfaz

- Clases y structs
- Herencia
- Métodos
- Polimorfismo
- Clases y miembros de clase abstractos y sellados
- Propiedades
- Eventos
- Indizadores

Vea también

- Guía de programación de C#
- Herencia
- Nombres de identificador

Métodos

18/03/2020 • 38 minutes to read • Edit Online

Un método es un bloque de código que contiene una serie de instrucciones. Un programa hace que se ejecuten las instrucciones al llamar al método y especificando los argumentos de método necesarios. En C#, todas las instrucciones ejecutadas se realizan en el contexto de un método. El método `Main` es el punto de entrada para cada aplicación de C# y se llama mediante Common Language Runtime (CLR) cuando se inicia el programa.

> **NOTE**
> En este tema se analizan los métodos denominados. Para obtener información sobre las funciones anónimas, vea Funciones anónimas (Guía de programación de C#).

Firmas de método

Los métodos se declaran en una `class` o `struct` al especificar:

- Un nivel de acceso opcional, como, por ejemplo, `public` o `private`. El valor predeterminado es `private`.
- Modificadores opcionales, como, por ejemplo, `abstract` o `sealed`.
- El valor devuelto o, si el método no tiene ninguno, `void`.
- El nombre del método.
- Los parámetros del método. Los parámetros de método se encierran entre paréntesis y se separan por comas. Los paréntesis vacíos indican que el método no requiere parámetros.

Todas estas partes forman la firma del método.

> **NOTE**
> Un tipo de valor devuelto de un método no forma parte de la firma del método con el objetivo de sobrecargar el método. Sin embargo, forma parte de la firma del método al determinar la compatibilidad entre un delegado y el método que señala.

En el siguiente ejemplo se define una clase denominada `Motorcycle` que contiene cinco métodos:

```
using System;

abstract class Motorcycle
{
   // Anyone can call this.
   public void StartEngine() {/* Method statements here */ }

   // Only derived classes can call this.
   protected void AddGas(int gallons) { /* Method statements here */ }

   // Derived classes can override the base class implementation.
   public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

   // Derived classes can override the base class implementation.
   public virtual int Drive(TimeSpan time, int speed) { /* Method statements here */ return 0; }

   // Derived classes must implement this.
   public abstract double GetTopSpeed();
}
```

Tenga en cuenta que la clase `Motorcycle` incluye un método sobrecargado, `Drive`. Dos métodos tienen el mismo nombre, pero se deben diferenciar en sus tipos de parámetros.

Invocación del método.

Los métodos pueden ser de *instancia* o *estáticos*. Para invocar un método de instancia es necesario crear una instancia de un objeto y llamar al método del objeto; el método de una instancia actúa en dicha instancia y sus datos. Si quiere invocar un método estático, haga referencia al nombre del tipo al que pertenece el método; los métodos estáticos no actúan en datos de instancia. Al intentar llamar a un método estático mediante una instancia de objeto se genera un error del compilador.

Llamar a un método es como acceder a un campo. Después del nombre de objeto (si se llama a un método de instancia) o el nombre de tipo (si llama a un método `static`), agregue un punto, el nombre del método y paréntesis. Los argumentos se enumeran entre paréntesis y están separados por comas.

La definición del método especifica los nombres y tipos de todos los parámetros necesarios. Cuando un autor de llamada invoca el método, proporciona valores concretos denominados argumentos para cada parámetro. Los argumentos deben ser compatibles con el tipo de parámetro, pero el nombre de argumento, si se usa alguno en el código de llamada, no tiene que ser el mismo que el del parámetro con nombre definido en el método. En el ejemplo siguiente, el método `Square` incluye un parámetro único de tipo `int` denominado *i*. La primera llamada de método pasa al método `Square` una variable de tipo `int` denominada *num*; la segunda, una constante numérica; y la tercera, una expresión.

```
public class Example
{
   public static void Main()
   {
      // Call with an int variable.
      int num = 4;
      int productA = Square(num);

      // Call with an integer literal.
      int productB = Square(12);

      // Call with an expression that evaluates to int.
      int productC = Square(productA * 3);
   }

   static int Square(int i)
   {
      // Store input argument in a local variable.
      int input = i;
      return input * input;
   }
}
```

La forma más común de invocación de método usa argumentos posicionales; proporciona argumentos en el mismo orden que los parámetros de método. Los métodos de la clase `Motorcycle` se pueden llamar como en el ejemplo siguiente. Por ejemplo, la llamada al método `Drive` incluye dos argumentos que se corresponden con los dos parámetros de la sintaxis del método. El primero se convierte en el valor del parámetro `miles` y el segundo en el valor del parámetro `speed`.

```
class TestMotorcycle : Motorcycle
{
   public override double GetTopSpeed()
   {
      return 108.4;
   }

   static void Main()
   {

      TestMotorcycle moto = new TestMotorcycle();

      moto.StartEngine();
      moto.AddGas(15);
      moto.Drive(5, 20);
      double speed = moto.GetTopSpeed();
      Console.WriteLine("My top speed is {0}", speed);
   }
}
```

También se pueden usar *argumentos con nombre* en lugar de argumentos posicionales al invocar un método. Cuando se usan argumentos con nombre, el nombre del parámetro se especifica seguido de dos puntos (":") y el argumento. Los argumentos del método pueden aparecer en cualquier orden, siempre que todos los argumentos necesarios están presentes. En el ejemplo siguiente se usan argumentos con nombre para invocar el método `TestMotorcycle.Drive`. En este ejemplo, los argumentos con nombre se pasan en orden inverso desde la lista de parámetros del método.

```
using System;

class TestMotorcycle : Motorcycle
{
   public override int Drive(int miles, int speed)
   {
      return (int) Math.Round( ((double)miles) / speed, 0);
   }

   public override double GetTopSpeed()
   {
      return 108.4;
   }

   static void Main()
   {

      TestMotorcycle moto = new TestMotorcycle();
      moto.StartEngine();
      moto.AddGas(15);
      var travelTime = moto.Drive(speed: 60, miles: 170);
      Console.WriteLine("Travel time: approx. {0} hours", travelTime);
   }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

Un método se puede invocar con argumentos posicionales y argumentos con nombre. Pero un argumento posicional no puede seguir a un argumento con nombre. En el ejemplo siguiente se invoca el método `TestMotorcycle.Drive` del ejemplo anterior con un argumento posicional y un argumento con nombre.

```
var travelTime = moto.Drive(170, speed: 55);
```

Métodos heredados e invalidados

Además de los miembros que se definen explícitamente en un tipo, un tipo hereda miembros definidos en sus clases base. Dado que todos los tipos en el sistema de tipo administrado heredan directa o indirectamente de la clase Object, todos los tipos heredan sus miembros, como Equals(Object), GetType() y ToString(). En el ejemplo siguiente se define una clase `Person`, se crean instancias de dos objetos `Person` y se llama al método `Person.Equals` para determinar si los dos objetos son iguales. Pero el método `Equals` no está definido en la clase `Person`; se hereda de Object.

```
using System;

public class Person
{
   public String FirstName;
}

public class Example
{
   public static void Main()
   {
      var p1 = new Person();
      p1.FirstName = "John";
      var p2 = new Person();
      p2.FirstName = "John";
      Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
   }
}
// The example displays the following output:
//      p1 = p2: False
```

Los tipos pueden invalidar miembros heredados usando la palabra clave `override` y proporcionando una implementación para el método invalidado. La firma del método debe ser igual a la del método invalidado. El ejemplo siguiente es similar al anterior, salvo que invalida el método Equals(Object). (También invalida el método GetHashCode(), ya que los dos métodos están diseñados para proporcionar resultados coherentes).

```
using System;

public class Person
{
   public String FirstName;

   public override bool Equals(object obj)
   {
      var p2 = obj as Person;
      if (p2 == null)
         return false;
      else
         return FirstName.Equals(p2.FirstName);
   }

   public override int GetHashCode()
   {
      return FirstName.GetHashCode();
   }
}

public class Example
{
   public static void Main()
   {
      var p1 = new Person();
      p1.FirstName = "John";
      var p2 = new Person();
      p2.FirstName = "John";
      Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
   }
}
// The example displays the following output:
//      p1 = p2: True
```

Pasar parámetros

Todos los tipos de C# son *tipos de valor* o *tipos de referencia*. Para obtener una lista de tipos de valor integrados, vea Tipos y variables. De forma predeterminada, los tipos de valor y los tipos de referencia se pasan a un método por valor.

Pasar parámetros por valor

Cuando un tipo de valor se pasa a un método por valor, se pasa una copia del objeto y no el propio objeto. Por lo tanto, los cambios realizados en el objeto en el método llamado no tienen ningún efecto en el objeto original cuando el control vuelve al autor de la llamada.

En el ejemplo siguiente se pasa un tipo de valor a un método por valor, y el método llamado intenta cambiar el valor del tipo de valor. Define una variable de tipo `int`, que es un tipo de valor, inicializa su valor en 20 y lo pasa a un método denominado `ModifyValue` que cambia el valor de la variable a 30. Pero cuando el método vuelve, el valor de la variable no cambia.

```
using System;

public class Example
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

Cuando un objeto de un tipo de referencia se pasa a un método por valor, se pasa por valor una referencia al objeto. Es decir, el método no recibe el objeto concreto, sino un argumento que indica la ubicación del objeto. Si cambia un miembro del objeto mediante esta referencia, el cambio se reflejará en el objeto cuando el control vuelva al método de llamada. Pero el reemplazo del objeto pasado al método no tendrá ningún efecto en el objeto original cuando el control vuelva al autor de la llamada.

En el ejemplo siguiente se define una clase (que es un tipo de referencia) denominada `SampleRefType`. Crea una instancia de un objeto `SampleRefType`, asigna 44 a su campo `value` y pasa el objeto al método `ModifyObject`. Fundamentalmente, este ejemplo hace lo mismo que el ejemplo anterior: pasa un argumento por valor a un método. Pero, debido a que se usa un tipo de referencia, el resultado es diferente. La modificación que se lleva a cabo en `ModifyObject` para el campo `obj.value` cambia también el campo `value` del argumento, `rt`, en el método `Main` a 33, tal y como muestra el resultado del ejemplo.

```
using System;

public class SampleRefType
{
    public int value;
}

public class Example
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.value = 44;
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj)
    {
        obj.value = 33;
    }
}
```

Pasar parámetros por referencia

Pase un parámetro por referencia cuando quiera cambiar el valor de un argumento en un método y reflejar ese cambio cuando el control vuelva al método de llamada. Para pasar un parámetro por referencia, use la palabra clave `ref` o `out`. También puede pasar un valor por referencia para evitar la copia, pero impedir modificaciones igualmente usando la palabra clave `in`.

El ejemplo siguiente es idéntico al anterior, salvo que el valor se pasa por referencia al método `ModifyValue`. Cuando se modifica el valor del parámetro en el método `ModifyValue`, el cambio del valor se refleja cuando el control vuelve al autor de la llamada.

```
using System;

public class Example
{
   public static void Main()
   {
      int value = 20;
      Console.WriteLine("In Main, value = {0}", value);
      ModifyValue(ref value);
      Console.WriteLine("Back in Main, value = {0}", value);
   }

   static void ModifyValue(ref int i)
   {
      i = 30;
      Console.WriteLine("In ModifyValue, parameter value = {0}", i);
      return;
   }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

Un patrón común que se usa en parámetros ref implica intercambiar los valores de variables. Se pasan dos variables a un método por referencia y el método intercambia su contenido. En el ejemplo siguiente se intercambian valores enteros.

```
using System;

public class Example
{
   static void Main()
   {
      int i = 2, j = 3;
      System.Console.WriteLine("i = {0}  j = {1}" , i, j);

      Swap(ref i, ref j);

      System.Console.WriteLine("i = {0}  j = {1}" , i, j);
   }

   static void Swap(ref int x, ref int y)
   {
      int temp = x;
      x = y;
      y = temp;
   }
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Pasar un parámetro de tipo de referencia le permite cambiar el valor de la propia referencia, en lugar del valor de sus campos o elementos individuales.

Matrices de parámetros

A veces, el requisito de especificar el número exacto de argumentos al método es restrictivo. El uso de la palabra clave `params` para indicar que un parámetro es una matriz de parámetros permite llamar al método con un número variable de argumentos. El parámetro etiquetado con la palabra clave `params` debe ser un tipo de matriz y ser el último parámetro en la lista de parámetros del método.

Un autor de llamada puede luego invocar el método de una de las tres maneras siguientes:

- Si se pasa una matriz del tipo adecuado que contenga el número de elementos que se quiera.
- Si se pasa una lista separada por comas de los argumentos individuales del tipo adecuado para el método.
- Si no se proporciona un argumento a la matriz de parámetros.

En el ejemplo siguiente se define un método denominado `GetVowels` que devuelve todas las vocales de una matriz de parámetros. El método `Main` muestra las tres formas de invocar el método. Los autores de llamadas no deben proporcionar argumentos para los parámetros que incluyen el modificador `params`. En ese caso, el parámetro es `null`.

```
using System;
using System.Linq;

class Example
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments: '{fromMultipleArguments}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { 'A', 'E', 'I', 'O', 'U' };
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter => vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from no value: ''
```

Argumentos y parámetros opcionales

La definición de un método puede especificar que sus parámetros son necesarios o que son opcionales. Los parámetros son necesarios de forma predeterminada. Para especificar parámetros opcionales se incluye el valor

predeterminado del parámetro en la definición del método. Cuando se llama al método, si no se proporciona ningún argumento para un parámetro opcional, se usa el valor predeterminado.

El valor predeterminado del parámetro debe asignarse con uno de los siguientes tipos de expresiones:

- Una constante, como una cadena literal o un número.
- Una expresión con el formato `new ValType()`, donde `ValType` es un tipo de valor. Tenga en cuenta que esta acción invoca el constructor sin parámetros implícito del tipo de valor, que no es un miembro real del tipo.
- Una expresión con el formato `default(ValType)`, donde `ValType` es un tipo de valor.

Si un método incluye parámetros necesarios y opcionales, los parámetros opcionales se definen al final de la lista de parámetros, después de todos los parámetros necesarios.

En el ejemplo siguiente se define un método, `ExampleMethod`, que tiene un parámetro necesario y dos opcionales.

```
using System;

public class Options
{
   public void ExampleMethod(int required, int optionalInt = default(int),
                             string description = "Optional Description")
   {
      Console.WriteLine("{0}: {1} + {2} = {3}", description, required,
                        optionalInt, required + optionalInt);
   }
}
```

Si se invoca un método con varios argumentos opcionales mediante argumentos posicionales, el autor de la llamada debe proporcionar un argumento para todos los parámetros opcionales, del primero al último, a los que se proporcione un argumento. Por ejemplo, en el caso del método `ExampleMethod`, si el autor de la llamada proporciona un argumento para el parámetro `description`, también debe proporcionar uno para el parámetro `optionalInt`. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` es una llamada de método válida; `opt.ExampleMethod(2, , "Addition of 2 and 0");` genera un error del compilador, "Falta un argumento".

Si se llama a un método mediante argumentos con nombre o una combinación de argumentos posicionales y con nombre, el autor de la llamada puede omitir los argumentos que siguen al último argumento posicional en la llamada al método.

En el ejemplo siguiente se llama tres veces al método `ExampleMethod`. Las dos primeras llamadas al método usan argumentos posicionales. La primera omite los dos argumentos opcionales, mientras que la segunda omite el último argumento. La tercera llamada al método proporciona un argumento posicional para el parámetro necesario, pero usa un argumento con nombre para proporcionar un valor al parámetro `description` mientras omite el argumento `optionalInt`.

```
public class Example
{
   public static void Main()
   {
      var opt = new Options();
      opt.ExampleMethod(10);
      opt.ExampleMethod(10, 2);
      opt.ExampleMethod(12, description: "Addition with zero:");
   }
}
// The example displays the following output:
//      Optional Description: 10 + 0 = 10
//      Optional Description: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

El uso de parámetros opcionales incide en la *resolución de sobrecarga* o en la manera en que el compilador de C# determina qué sobrecarga en particular debe invocar una llamada al método, como sigue:

- Un método, indexador o constructor es un candidato para la ejecución si cada uno de sus parámetros es opcional o corresponde, por nombre o por posición, a un solo argumento de la instrucción de llamada y el argumento se puede convertir al tipo del parámetro.
- Si se encuentra más de un candidato, se aplican las reglas de resolución de sobrecarga de las conversiones preferidas a los argumentos que se especifican explícitamente. Los argumentos omitidos en parámetros opcionales se ignoran.
- Si dos candidatos se consideran igualmente correctos, la preferencia pasa a un candidato que no tenga parámetros opcionales cuyos argumentos se hayan omitido en la llamada. Se trata de una consecuencia de una preferencia general en la resolución de sobrecarga para los candidatos con menos parámetros.

Valores devueltos

Los métodos pueden devolver un valor al autor de llamada. Si el tipo de valor devuelto (el tipo que aparece antes del nombre de método) no es `void`, el método puede devolver el valor mediante la palabra clave `return`. Una instrucción con la palabra clave `return` seguida de una variable, una constante o una expresión que coincide con el tipo de valor devuelto devolverá este valor al autor de la llamada al método. Los métodos con un tipo de valor devuelto no nulo son necesarios para usar la palabra clave `return` para devolver un valor. La palabra clave `return` también detiene la ejecución del método.

Si el tipo de valor devuelto es `void`, una instrucción `return` sin un valor también es útil para detener la ejecución del método. Sin la palabra clave `return` , el método dejará de ejecutarse cuando alcance el final del bloque de código.

Por ejemplo, estos dos métodos utilizan la palabra clave `return` para devolver enteros:

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Para utilizar un valor devuelto de un método, el método de llamada puede usar la llamada de método en cualquier lugar; un valor del mismo tipo sería suficiente. También puede asignar el valor devuelto a una variable. Por ejemplo, los dos siguientes ejemplos de código logran el mismo objetivo:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Usar una variable local, en este caso, `result`, para almacenar un valor es opcional. La legibilidad del código puede

ser útil, o puede ser necesaria si debe almacenar el valor original del argumento para todo el ámbito del método.

A veces, quiere que el método devuelva más que un solo valor. A partir de C# 7.0, puede hacer esto fácilmente mediante *tipos de tupla* y *literales de tupla*. El tipo de tupla define los tipos de datos de los elementos de la tupla. Los literales de tupla proporcionan los valores reales de la tupla devuelta. En el ejemplo siguiente, `(string, string, string, int)` define el tipo de tupla que devuelve el método `GetPersonalInfo`. La expresión `(per.FirstName, per.MiddleName, per.LastName, per.Age)` es el literal de tupla; el método devuelve el nombre, los apellidos y la edad de un objeto `PersonInfo`.

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

Luego, el autor de la llamada puede usar la tupla devuelta con código como el siguiente:

```
var person = GetPersonalInfo("111111111")
Console.WriteLine("{person.Item1} {person.Item3}: age = {person.Item4}");
```

También se pueden asignar nombres a los elementos de tupla en la definición de tipo de tupla. En el ejemplo siguiente se muestra una versión alternativa del método `GetPersonalInfo` que usa elementos con nombre:

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

La llamada anterior al método `GetPersonInfo` se puede modificar luego de la manera siguiente:

```
var person = GetPersonalInfo("111111111");
Console.WriteLine("{person.FName} {person.LName}: age = {person.Age}");
```

Si a un método se pasa una matriz como argumento y modifica el valor de elementos individuales, no es necesario que el método devuelva la matriz, aunque puede que se prefiera hacerlo a efectos del buen estilo o el flujo funcional de valores. Esto se debe a que C# pasa todos los tipos de referencia por valor, y el valor de una referencia a la matriz es el puntero a la matriz. En el ejemplo siguiente, los cambios al contenido de la matriz `values` que se realizan en el método `DoubleValues` los puede observar cualquier código que tenga una referencia a la matriz.

```
using System;

public class Example
{
   static void Main(string[] args)
   {
      int[] values = { 2, 4, 6, 8 };
      DoubleValues(values);
      foreach (var value in values)
         Console.Write("{0}  ", value);
   }

   public static void DoubleValues(int[] arr)
   {
      for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
         arr[ctr] = arr[ctr] * 2;
   }
}
// The example displays the following output:
//       4  8  12  16
```

Métodos de extensión

Normalmente, hay dos maneras de agregar un método a un tipo existente:

- Modificar el código fuente del tipo. No puede hacerlo si no es propietario del código fuente del tipo. Y esto supone un cambio sustancial si también agrega los campos de datos privados para admitir el método.
- Definir el nuevo método en una clase derivada. No se puede agregar un método de este modo con herencia para otros tipos, como estructuras y enumeraciones. Tampoco se puede usar para agregar un método a una clase sealed.

Los métodos de extensión permiten agregar un método a un tipo existente sin modificar el propio tipo o implementar el nuevo método en un tipo heredado. El método de extensión tampoco tiene que residir en el mismo ensamblado que el tipo que extiende. Llame a un método de extensión como si fuera miembro de un tipo definido.

Para más información, vea Métodos de extensión.

Métodos asincrónicos

Mediante la característica asincrónica, puede invocar métodos asincrónicos sin usar definiciones de llamada explícitas ni dividir manualmente el código en varios métodos o expresiones lambda.

Si marca un método con el modificador async , puede usar el operador await en el método. Cuando el control llega a una expresión `await` en el método asincrónico, el control se devuelve al autor de la llamada si la tarea en espera no se ha completado y se suspende el progreso del método con la palabra clave `await` hasta que dicha tarea se complete. Cuando se completa la tarea, la ejecución puede reanudarse en el método.

> **NOTE**
> Un método asincrónico vuelve al autor de llamada cuando encuentra el primer objeto esperado que aún no se ha completado o cuando llega al final del método asincrónico, lo que ocurra primero.

Un método asincrónico puede tener un tipo de valor devuelto de Task<TResult>, Task o `void`. El tipo de valor devuelto `void` se usa principalmente para definir controladores de eventos, donde se requiere un tipo de valor devuelto `void`. No se puede esperar un método asincrónico que devuelve `void` y el autor de llamada a un método que no devuelve ningún valor no puede capturar ninguna excepción producida por este. A partir de la

versión C# 7.0, el método asincrónico puede tener cualquier tipo de valor devuelto similar a una tarea .

En el ejemplo siguiente, `DelayAsync` es un método asincrónico que contiene una instrucción return que devuelve un entero. Como se trata de un método asincrónico, su declaración de método debe tener un tipo de valor devuelto de `Task<int>`. Dado que el tipo de valor devuelto es `Task<int>`, la evaluación de la expresión `await` en `DoSomethingAsync` genera un entero, como se demuestra en la instrucción `int result = await delayTask` siguiente.

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;

public class Example
{
    // This Click event is marked with the async modifier.
    public static void Main()
    {
       DoSomethingAsync().Wait();
    }

    private static async Task DoSomethingAsync()
    {
        int result = await DelayAsync();
        Console.WriteLine("Result: " + result);
    }

    private static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }

    // Output:
    //  Result: 5
}
// The example displays the following output:
//        Result: 5
```

Un método asincrónico no puede declarar ningún parámetro in, ref o out, pero puede llamar a los métodos que tienen estos parámetros.

Para obtener más información sobre los métodos asincrónicos, vea Programación asincrónica con Async y Await, Controlar el flujo en los programas asincrónicos y Tipos de valor devueltos asincrónicos.

Miembros con forma de expresión

Es habitual tener definiciones de método que simplemente hacen las devoluciones de forma inmediata con el resultado de una expresión, o que tienen una sola instrucción como cuerpo del método. Hay un acceso directo de sintaxis para definir este método mediante `=>`:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Si el método devuelve `void` o se trata de un método asincrónico, el cuerpo del método debe ser una expresión de instrucción (igual que con las expresiones lambda). Para propiedades e indexadores, deben ser de solo lectura, y no se usa la palabra clave de descriptor de acceso `get`.

Iterators

Un iterador realiza una iteración personalizada en una colección, como una lista o matriz. Un iterador usa la instrucción yield return para devolver cada elemento de uno en uno. Cuando se llega a una instrucción `yield return`, se recuerda la ubicación actual para que el autor de la llamada pueda solicitar el siguiente elemento en la secuencia.

El tipo de valor devuelto de un iterador puede ser IEnumerable, IEnumerable<T>, IEnumerator o IEnumerator<T>.

Para obtener más información, consulta Iteradores.

Vea también

- Modificadores de acceso
- Clases estáticas y sus miembros
- Herencia
- Clases y miembros de clase abstractos y sellados
- params
- out
- ref
- in
- Pasar parámetros

Expresiones lambda (Guía de programación de C#)

29/06/2020 • 20 minutes to read • Edit Online

Una *expresión lambda* es una expresión que tiene cualquiera de estas dos formas:

- Una lambda de expresión que tiene una expresión como cuerpo:

```
(input-parameters) => expression
```

- Una lambda de instrucción que tiene un bloque de instrucciones como cuerpo:

```
(input-parameters) => { <sequence-of-statements> }
```

Use el operador de declaración lambda `=>` para separar la lista de parámetros de la lamba de su cuerpo. Para crear una expresión lambda, especifique los parámetros de entrada (si existen) a la izquierda del operador lambda y una expresión o bloque de instrucciones en el otro lado.

Toda expresión lambda se puede convertir en un tipo delegado. El tipo delegado al que se puede convertir una expresión lambda se define según los tipos de sus parámetros y el valor devuelto. Si una expresión lambda no devuelve un valor, se puede convertir en uno de los tipos delegados `Action`; de lo contrario, se puede convertir en uno de los tipos delegados `Func`. Por ejemplo, una expresión lambda que tiene dos parámetros y no devuelve ningún valor corresponde a un delegado Action<T1,T2>. Una expresión lambda que tiene un parámetro y devuelve un valor se puede convertir en un delegado Func<T,TResult>. En el ejemplo siguiente, la expresión lambda `x => x * x`, que especifica un parámetro denominado `x` y devuelve el valor de `x` al cuadrado, se asigna a una variable de un tipo delegado:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Las expresiones lambda también se pueden convertir en los tipos de árbol de expresión, como se muestra en los ejemplos siguientes:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

Puede usar expresiones lambda en cualquier código que requiera instancias de tipos delegados o de árboles de expresión, por ejemplo, como un argumento del método Task.Run(Action) para pasar el código que se debe ejecutar en segundo plano. También puede usar expresiones lambda al escribir LINQ en C#, como se muestra en el ejemplo siguiente:

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

Cuando se usa la sintaxis de método para llamar al método Enumerable.Select en la clase System.Linq.Enumerable, por ejemplo en LINQ to Objects y en LINQ to XML, el parámetro es un tipo delegado System.Func<T,TResult>. Cuando se llama al método Queryable.Select en la clase System.Linq.Queryable, por ejemplo en LINQ to SQL, el tipo de parámetro es un tipo de árbol de expresión `Expression<Func<TSource,TResult>>`. En ambos casos, se puede usar la misma expresión lambda para especificar el valor del parámetro. Esto hace que las dos llamadas `Select` tengan un aspecto similar aunque, de hecho, el tipo de objetos creados a partir las lambdas es distinto.

Lambdas de expresión

Una expresión lambda con una expresión en el lado derecho del operador `=>` se denomina *lambda de expresión*. Las lambdas de expresión se usan ampliamente en la construcción de árboles de expresión. Una expresión lambda devuelve el resultado de evaluar la condición y tiene la siguiente forma:

```
(input-parameters) => expression
```

Los paréntesis solo son opcionales si la lambda tiene un parámetro de entrada; de lo contrario, son obligatorios.

Para especificar cero parámetros de entrada, utilice paréntesis vacíos:

```
Action line = () => Console.WriteLine();
```

Dos o más parámetros de entrada se separan por comas y se encierran entre paréntesis:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

A veces resulta imposible que el compilador deduzca los tipos de entrada. Puede especificar los tipos de manera explícita, tal como se muestra en el ejemplo siguiente:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Los tipos de parámetro de entrada deben ser todos explícitos o todos implícitos; de lo contrario, se produce un error del compilador CS0748.

El cuerpo de una expresión lambda puede constar de una llamada al método. Sin embargo, si crea árboles de expresión que se evalúan fuera del contexto de Common Language Runtime de .NET, como en SQL Server, no debe utilizar llamadas a métodos en expresiones lambda. Los métodos no tendrán ningún significado fuera del contexto de .NET Common Language Runtime.

Lambdas de instrucción

Una lambda de instrucción es similar a un lambda de expresión, salvo que las instrucciones se encierran entre llaves:

```
(input-parameters) => { <sequence-of-statements> }
```

El cuerpo de una lambda de instrucción puede estar compuesto de cualquier número de instrucciones; sin embargo, en la práctica, generalmente no hay más de dos o tres.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Las expresiones lambdas no se pueden usar para crear árboles de expresión.

Lambdas asincrónicas

Puede crear fácilmente expresiones e instrucciones lambda que incorporen el procesamiento asincrónico mediante las palabras clave async y await . Por ejemplo, en el siguiente ejemplo de formularios Windows Forms se incluye un controlador de eventos que llama y espera un método asincrónico, `ExampleMethodAsync`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Puede agregar el mismo controlador de eventos utilizando una lambda asincrónica. Para agregar este controlador, agregue un modificador `async` antes de la lista de parámetros lambda, como se muestra en el ejemplo siguiente:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Para obtener más información sobre cómo crear y usar métodos asincrónicos, vea Programación asincrónica con

async y await.

Expresiones lambda y tuplas

A partir de C# 7.0, el lenguaje C# proporciona compatibilidad integrada para las tuplas. Puede proporcionar una tupla como argumento a una expresión lambda, mientras que la expresión lambda también puede devolver una tupla. En algunos casos, el compilador de C# usa la inferencia de tipos para determinar los tipos de componentes de la tupla.

Para definir una tupla, incluya entre paréntesis una lista delimitada por comas de los componentes. En el ejemplo siguiente se usa la tupla con tres componentes para pasar una secuencia de números a una expresión lambda, que duplica cada valor y devuelve una tupla con tres componentes que contiene el resultado de las multiplicaciones.

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

Normalmente, los campos de una tupla se denominan `Item1`, `Item2`, etc., aunque puede definir una tupla con componentes con nombre, como en el ejemplo siguiente.

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

Para más información sobre las tuplas de C#, consulte el artículo sobre los tipos de tuplas de C#.

Lambdas con los operadores de consulta estándar

LINQ to Objects, entre otras implementaciones, tiene un parámetro de entrada cuyo tipo es uno de la familia Func<TResult> de delegados genéricos. Estos delegados usan parámetros de tipo para definir el número y el tipo de los parámetros de entrada, así como el tipo de valor devuelto del delegado. Los delegados `Func` son muy útiles para encapsular expresiones definidas por el usuario que se aplican a cada elemento en un conjunto de datos de origen. Por ejemplo, considere el tipo delegado Func<T,TResult>:

```
public delegate TResult Func<in T, out TResult>(T arg)
```

Se pueden crear instancias del delegado como una instancia `Func<int, bool>`, donde `int` es un parámetro de entrada y `bool` es el valor devuelto. El valor devuelto siempre se especifica en el último parámetro de tipo. Por ejemplo, `Func<int, string, bool>` define un delegado con dos parámetros de entrada, `int` y `string`, y un tipo de valor devuelto de `bool`. El delegado `Func` siguiente, cuando se invoca, devuelve un valor booleano que indica si el parámetro de entrada es igual a cinco:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

También puede proporcionar una expresión lambda cuando el tipo de argumento es Expression<TDelegate>, (por ejemplo, en los operadores de consulta estándar que se definen en el tipo Queryable). Al especificar un argumento Expression<TDelegate>, la lambda se compila en un árbol de expresión.

En el ejemplo siguiente se usa el operador de consulta estándar Count:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

El compilador puede deducir el tipo del parámetro de entrada o también se puede especificar explícitamente. Esta expresión lambda particular cuenta aquellos enteros (`n`) que divididos por dos dan como resto 1.

El siguiente ejemplo genera una secuencia que contiene todos los elementos de la matriz `numbers` que preceden al 9, ya que ese es el primer número de la secuencia que no cumple la condición:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

En el siguiente ejemplo se especifican varios parámetros de entrada encerrándolos entre paréntesis. El método devuelve todos los elementos de la matriz `numbers` hasta que encuentra un número cuyo valor es menor que la posición ordinal en la matriz:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Inferencia de tipos en expresiones lambda

Al escribir lambdas, no tiene por qué especificar un tipo para los parámetros de entrada, ya que el compilador puede deducir el tipo según el cuerpo de lambda, los tipos de parámetros y otros factores, tal como se describe en la especificación del lenguaje C#. Para la mayoría de los operadores de consulta estándar, la primera entrada es el tipo de los elementos en la secuencia de origen. Si está realizando una consulta sobre `IEnumerable<Customer>`, se deducirá que la variable de entrada será un objeto `Customer`, lo cual significa que se podrá tener acceso a sus métodos y propiedades:

```
customers.Where(c => c.City == "London");
```

Las reglas generales para la inferencia de tipos de las lambdas son las siguientes:

- La lambda debe contener el mismo número de parámetros que el tipo delegado.
- Cada parámetro de entrada de la lambda debe poder convertirse implícitamente a su parámetro de delegado correspondiente.
- El valor devuelto de la lambda (si existe) debe poder convertirse implícitamente al tipo de valor devuelto del delegado.

Observe que las expresiones lambda, en sí mismas, no tienen tipo, ya que el sistema de tipos comunes no tiene ningún concepto intrínseco de "expresión lambda". Sin embargo, a veces resulta práctico hablar coloquialmente del "tipo" de una expresión lambda. En estos casos, el tipo hace referencia al tipo del delegado o el tipo de Expression en el que se convierte la expresión lambda.

Captura de variables externas y el ámbito de las variables en las expresiones lambda

Las operaciones lambda pueden hacer referencia a *variables externas*. Estas son las variables que están en el ámbito del método que define la expresión lambda o en el ámbito del tipo que contiene la expresión lambda. Las variables que se capturan de esta manera se almacenan para su uso en la expresión lambda, cuando de otro modo quedarían fuera de ámbito y serían eliminadas por la recolección de elementos no utilizados. Para poder utilizar una variable externa en una expresión lambda, debe estar previamente asignada. En el ejemplo siguiente se muestran estas reglas:

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable: {equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

Las reglas siguientes se aplican al ámbito de las variables en las expresiones lambda:

- Una variable capturada no se recolectará como elemento no utilizado hasta que el delegado que hace referencia a ella sea elegible para la recolección de elementos no utilizados.
- Las variables introducidas en una expresión lambda no son visibles en el método envolvente.
- Una expresión lambda no puede capturar directamente un parámetro in, ref ni out desde el método envolvente.
- Una instrucción return en una expresión lambda no hace que el método envolvente devuelva un valor.
- Una expresión lambda no puede contener una instrucción goto, break ni continue si el destino de esa instrucción de salto está fuera del bloque de la expresión lambda. También es un error utilizar una instrucción de salto fuera del bloque de la expresión lambda si el destino está dentro del bloque.

Especificación del lenguaje C#

Para obtener más información, vea la sección Expresiones de función anónima de la Especificación del lenguaje C#.

Capítulo destacado del libro

Delegates, Events, and Lambda Expressions (Delegados, eventos y expresiones lambda) en C# 3.0 Cookbook, Tercera edición: More than 250 solutions for C# 3.0 programmers (Más de 250 soluciones para programadores de C# 3.0)

Vea también

- Guía de programación de C#
- LINQ (Language Integrated Query)
- Árboles de expresión
- Funciones locales frente a expresiones lambda
- Ejemplos de C# de Visual Studio 2008 (vea los archivos de ejemplos de consultas LINQ y el programa XQuery)
- Expresiones lambda recursivas

Propiedades

18/03/2020 • 17 minutes to read • Edit Online

Las propiedades son ciudadanos de primera clase en C#. El lenguaje define la sintaxis que permite a los desarrolladores escribir código que exprese con precisión su intención de diseño.

Las propiedades se comportan como campos cuando se obtiene acceso a ellas. Pero, a diferencia de los campos, las propiedades se implementan con descriptores de acceso que definen las instrucciones que se ejecutan cuando se tiene acceso a una propiedad o se asigna.

Sintaxis de las propiedades

La sintaxis para propiedades es una extensión natural de los campos. Un campo define una ubicación de almacenamiento:

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

Una definición de propiedad contiene las declaraciones para un descriptor de acceso `get` y `set` que recupera y asigna el valor de esa propiedad:

```
public class Person
{
    public string FirstName { get; set; }

    // remaining implementation removed from listing
}
```

La sintaxis anterior es la sintaxis *de propiedades automáticas*. El compilador genera la ubicación de almacenamiento para el campo que respalda a la propiedad. El compilador también implementa el cuerpo de los descriptores de acceso `get` y `set`.

A veces, necesita inicializar una propiedad en un valor distinto del predeterminado para su tipo. C# permite esto estableciendo un valor después de la llave de cierre de la propiedad. Puede que prefiera que el valor inicial para la propiedad `FirstName` sea la cadena vacía en lugar de `null`. Debe especificarlo como se muestra a continuación:

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

La inicialización específica es más útil en las propiedades de solo lectura, como verá posteriormente en este artículo.

También puede definir su propio almacenamiento, como se muestra a continuación:

```
public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

Cuando una implementación de propiedad es una expresión única, puede usar *miembros con forma de expresión* para el captador o establecedor:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

Esta sintaxis simplificada se usará cuando sea necesario en todo el artículo.

La definición de propiedad anterior es una propiedad de lectura y escritura. Observe la palabra clave `value` en el descriptor de acceso set. El descriptor de acceso `set` siempre tiene un único parámetro denominado `value`. El descriptor de acceso `get` tiene que devolver un valor que se pueda convertir al tipo de la propiedad (`string` en este ejemplo).

Estos son los conceptos básicos de la sintaxis. Hay muchas variantes distintas que admiten diversos lenguajes de diseño diferentes. Vamos a explorarlas y a aprender las opciones de sintaxis de cada una.

Escenarios

Los ejemplos anteriores mostraron uno de los casos más simples de definición de propiedad: una propiedad de lectura y escritura sin validación. Al escribir el código que quiere en los descriptores de acceso `get` y `set`, puede crear muchos escenarios diferentes.

Validación

Puede escribir código en el descriptor de acceso `set` para asegurarse de que los valores representados por una propiedad siempre son válidos. Por ejemplo, suponga que una regla para la clase `Person` es que el nombre no puede estar en blanco ni tener espacios en blanco. Se escribiría de esta forma:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            firstName = value;
        }
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

El ejemplo anterior se puede simplificar usando una expresión `throw` como parte de la validación del establecedor de propiedad:

```
public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = (!string.IsNullOrWhiteSpace(value)) ? value : throw new ArgumentException("First
name must not be blank");
    }
    private string firstName;
    // remaining implementation removed from listing
}
```

En el ejemplo anterior, se aplica la regla de que el nombre no debe estar en blanco ni tener espacios en blanco. Si un desarrollador escribe:

```
hero.FirstName = "";
```

Esa asignación produce una excepción `ArgumentException`. Dado que un descriptor de acceso set de propiedad debe tener un tipo de valor devuelto void, los errores se notifican en el descriptor de acceso set iniciando una excepción.

Se puede extender esta misma sintaxis para todo lo que se necesite en el escenario. Se pueden comprobar las relaciones entre las diferentes propiedades o validar con respecto a cualquier condición externa. Todas las instrucciones de C# válidas son válidas en un descriptor de acceso de propiedad.

De sólo lectura

Hasta ahora, todas las definiciones de propiedad que se vieron son propiedades de lectura y escritura con descriptores de acceso públicos. No es la única accesibilidad válida para las propiedades. Se pueden crear propiedades de solo lectura, o proporcionar accesibilidad diferente a los descriptores de acceso set y get. Suponga que su clase `Person` solo debe habilitar el cambio del valor de la propiedad `FirstName` desde otros métodos de esa clase. Podría asignar al descriptor de acceso set la accesibilidad `private` en lugar de `public`:

```
public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}
```

Ahora, se puede obtener acceso a la propiedad `FirstName` desde cualquier código, pero solo puede asignarse desde otro código de la clase `Person`.

Puede agregar cualquier modificador de acceso restrictivo al descriptor de acceso set o get. Ningún modificador de acceso que se coloque en el descriptor de acceso concreto debe ser más limitado que el modificador de acceso en la definición de la propiedad. El ejemplo anterior es válido porque la propiedad `FirstName` es `public`, pero el descriptor de acceso set es `private`. No se puede declarar una propiedad `private` con un descriptor de acceso `public`. Las declaraciones de propiedad también se pueden declarar como `protected`, `internal`, `protected internal` o incluso `private`.

También es válido colocar el modificador más restrictivo en el descriptor de acceso `get`. Por ejemplo, se podría tener una propiedad `public`, pero restringir el descriptor de acceso `get` a `private`. Ese escenario raramente se aplica en la práctica.

También puede restringir las modificaciones de una propiedad, de manera que solo pueda establecerse en un constructor o en un inicializador de propiedades. Puede modificar la clase `Person` de la manera siguiente:

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

Esta característica se usa normalmente para inicializar colecciones que están expuestas como propiedades de solo lectura:

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
}
```

Propiedades calculadas

Una propiedad no tiene por qué devolver únicamente el valor de un campo de miembro. Se pueden crear propiedades que devuelvan un valor calculado. Vamos a ampliar el objeto `Person` para que devuelva el nombre completo, que se calcula mediante la concatenación del nombre y el apellido:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

En el ejemplo anterior se usa la característica de interpolación de cadenas para crear la cadena con formato para el nombre completo.

También se pueden usar un *miembro con forma de expresión*, que proporciona una manera más concisa de crear la propiedad `FullName` calculada:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Los *miembros con forma de expresión* usan la sintaxis de *expresión lambda* para definir métodos que contienen una única expresión. En este caso, esa expresión devuelve el nombre completo para el objeto person.

Propiedades de evaluación en caché

Se puede combinar el concepto de una propiedad calculada con almacenamiento de información y crear una *propiedad de evaluación en caché*. Por ejemplo, se podría actualizar la propiedad `FullName` para que la cadena de formato solo apareciera la primera vez que se obtuvo acceso a ella:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

Pero el código anterior contiene un error. Si el código actualiza el valor de la propiedad `FirstName` o `LastName`, el campo evaluado previamente `fullName` no es válido. Hay que modificar los descriptores de acceso `set` de la propiedad `FirstName` y `LastName` para que el campo `fullName` se calcule de nuevo:

```
public class Person
{
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            firstName = value;
            fullName = null;
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        set
        {
            lastName = value;
            fullName = null;
        }
    }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == null)
                fullName = $"{FirstName} {LastName}";
            return fullName;
        }
    }
}
```

Esta versión final da como resultado la propiedad `FullName` solo cuando sea necesario. Si la versión calculada previamente es válida, es la que se usa. Si otro cambio de estado invalida la versión calculada previamente, se vuelve a calcular. No es necesario que los desarrolladores que usan esta clase conozcan los detalles de la implementación. Ninguno de estos cambios internos afectan al uso del objeto Person. Es el motivo principal para usar propiedades para exponer los miembros de datos de un objeto.

Asociar atributos a propiedades implementadas automáticamente

A partir de C# 7.3, los atributos de campo se pueden conectar al campo de respaldo generado por el compilador en las propiedades implementadas automáticamente. Por ejemplo, pensemos en una revisión de la clase `Person` que agrega una propiedad `Id` de entero único. Escribimos la propiedad `Id` usando una propiedad implementada automáticamente, pero el diseño no requiere que la propiedad `Id` se conserve. NonSerializedAttribute solo se puede asociar a campos, no a propiedades. NonSerializedAttribute se puede asociar al campo de respaldo de la propiedad `Id` usando el especificador `field:` en el atributo, como se muestra en el siguiente ejemplo:

```
public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

Esta técnica funciona con cualquier atributo que se asocie al campo de respaldo en la propiedad implementada automáticamente.

Implementar INotifyPropertyChanged

Un último escenario donde se necesita escribir código en un descriptor de acceso de propiedad es para admitir la interfaz INotifyPropertyChanged que se usa para notificar a los clientes de enlace de datos el cambio de un valor. Cuando se cambia el valor de una propiedad, el objeto genera el evento INotifyPropertyChanged.PropertyChanged para indicar el cambio. A su vez, las bibliotecas de enlace de datos actualizan los elementos de visualización en función de ese cambio. El código siguiente muestra cómo se implementaría `INotifyPropertyChanged` para la propiedad `FirstName` de esta clase person.

```
public class Person : INotifyPropertyChanged
{
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != firstName)
            {
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
            firstName = value;
        }
    }
    private string firstName;

    public event PropertyChangedEventHandler PropertyChanged;
    // remaining implementation removed from listing
}
```

El operador `?.` se denomina *operador condicional NULL*. Comprueba si existe una referencia nula antes de evaluar el lado derecho del operador. El resultado final es que si no hay ningún suscriptor para el evento `PropertyChanged`, no se ejecuta el código para generar el evento. En ese caso, se producirá una `NullReferenceException` sin esta comprobación. Para obtener más información, vea `events`. En este ejemplo también se usa el nuevo operador `nameof` para convertir el símbolo de nombre de propiedad en su representación de texto. Con `nameof` se pueden reducir los errores en los que no se escribió correctamente el nombre de la propiedad.

De nuevo, la implementación de INotifyPropertyChanged es un ejemplo de un caso en el que se puede escribir código en los descriptores de acceso para admitir los escenarios que se necesitan.

Resumen

Las propiedades son una forma de campos inteligentes en una clase o un objeto. Desde fuera del objeto, parecen campos en el objeto. Pero las propiedades pueden implementarse mediante la paleta completa de funcionalidad de C#. Se puede proporcionar validación, tipos diferentes de accesibilidad, evaluación diferida o los requisitos que se necesiten para cada escenario.

Indizadores

29/06/2020 • 15 minutes to read • Edit Online

Los *indizadores* son similares a las propiedades. Muchas veces, los indizadores se basan en las mismas características del lenguaje que las propiedades. Los indizadores permiten las propiedades *indizadas*: propiedades a las que se hace referencia mediante uno o más argumentos. Estos argumentos proporcionan un índice en alguna colección de valores.

Sintaxis del indizador

Se obtiene acceso a un indizador usando un nombre de variable y corchetes. Los argumentos del indizador se colocan dentro de los corchetes:

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

Los indizadores se declaran con la palabra clave `this` como nombre de la propiedad y declarando los argumentos entre corchetes. Esta declaración coincidiría con el uso que se muestra en el párrafo anterior:

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

En este ejemplo inicial puede ver la relación existente entre la sintaxis de las propiedades y los indizadores. Esta analogía lleva a cabo la mayoría de las reglas de sintaxis de los indizadores. Los indizadores pueden tener cualquier modificador de acceso válido (público, interno protegido, protegido, interno, privado o privado protegido). Pueden ser sellados, virtuales o abstractos. Al igual que con las propiedades, puede especificar distintos modificadores de acceso para los descriptores de acceso get y set en un indizador. También puede especificar indizadores de solo lectura (omitiendo el descriptor de acceso set) o indizadores de solo escritura (omitiendo el descriptor de acceso get).

Puede aplicar a los indizadores casi todo lo que aprenda al trabajar con propiedades. La única excepción a esta regla son las *propiedades implementadas automáticamente*. El compilador no siempre puede generar el almacenamiento correcto para un indizador.

La presencia de argumentos para hacer referencia a un elemento en un conjunto de elementos distingue los indizadores de las propiedades. Puede definir varios indizadores en un tipo, mientras que las listas de argumentos de cada indizador son únicas. Vamos a explorar escenarios diferentes en los que puede usar uno o varios indizadores en una definición de clase.

Escenarios

Tendría que definir *indizadores* en el tipo si su API modela alguna colección en la que se definen los argumentos de esa colección. Los indizadores pueden (o no) asignarse directamente a los tipos de colección que forman parte del marco de trabajo principal de .NET. El tipo puede tener otras responsabilidades, además de tener que modelar una colección. Los indizadores le permiten proporcionar la API que coincida con la abstracción de su tipo sin tener que exponer la información interna de cómo se almacenan o se calculan los valores de dicha abstracción.

Veamos algunos de los escenarios habituales en los que se usan los *indizadores*. Puede obtener acceso a la carpeta de ejemplo para indexadores. Para obtener instrucciones de descarga, vea Ejemplos y tutoriales.

Matrices y vectores

Uno de los escenarios más comunes para crear indizadores es cuando el tipo modela una matriz o un vector. Puede crear un indizador para modelar una lista ordenada de datos.

La ventaja de crear su propio indizador es que puede definir el almacenamiento de esa colección para satisfacer sus necesidades. Piense en un escenario en el que el tipo modela datos históricos que tienen un tamaño demasiado grande para poder cargarlos en la memoria de una vez. Debe cargar y descargar secciones de la colección en función del uso. En el ejemplo siguiente se modela este comportamiento. Informa sobre el número de puntos de datos existente, crea páginas para incluir secciones de los datos a petición y quita páginas de la memoria para dejar espacio para las páginas necesarias para las solicitudes más recientes.

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new List<Measurements>();
        private readonly int startingIndex;
        private readonly int length;
        private bool dirty;
        private DateTime lastAccess;

        public Page(int startingIndex, int length)
        {
            this.startingIndex = startingIndex;
            this.length = length;
            lastAccess = DateTime.Now;

            // This stays as random stuff:
            var generator = new Random();
            for(int i=0; i < length; i++)
            {
                var m = new Measurements
                {
                    HiTemp = generator.Next(50, 95),
                    LoTemp = generator.Next(12, 49),
                    AirPressure = 28.0 + generator.NextDouble() * 4
                };
                pageData.Add(m);
            }
        }
        public bool HasItem(int index) =>
            ((index >= startingIndex) &&
            (index < startingIndex + length));

        public Measurements this[int index]
        {
            get
            {
                lastAccess = DateTime.Now;
                return pageData[index - startingIndex];
            }
            set
            {
                pageData[index - startingIndex] = value;
                dirty = true;
                lastAccess = DateTime.Now;
            }
        }

        public bool Dirty => dirty;
        public DateTime LastAccess => lastAccess;
    }
```

```
    private readonly int totalSize;
    private readonly List<Page> pagesInMemory = new List<Page>();

    public DataSamples(int totalSize)
    {
        this.totalSize = totalSize;
    }

    public Measurements this[int index]
    {
        get
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");

            var page = updateCachedPagesForAccess(index);
            return page[index];
        }
        set
        {
            if (index < 0)
                throw new IndexOutOfRangeException("Cannot index less than 0");
            if (index >= totalSize)
                throw new IndexOutOfRangeException("Cannot index past the end of storage");
            var page = updateCachedPagesForAccess(index);

            page[index] = value;
        }
    }

    private Page updateCachedPagesForAccess(int index)
    {
        foreach (var p in pagesInMemory)
        {
            if (p.HasItem(index))
            {
                return p;
            }
        }
        var startingIndex = (index / 1000) * 1000;
        var newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    private void addPageToCache(Page p)
    {
        if (pagesInMemory.Count > 4)
        {
            // remove oldest non-dirty page:
            var oldest = pagesInMemory
                .Where(page => !page.Dirty)
                .OrderBy(page => page.LastAccess)
                .FirstOrDefault();
            // Note that this may keep more than 5 pages in memory
            // if too much is dirty
            if (oldest != null)
                pagesInMemory.Remove(oldest);
        }
        pagesInMemory.Add(p);
    }
}
```

Puede seguir esta expresión de diseño para modelar cualquier tipo de colección cuando haya motivos de peso para

no cargar todo el conjunto de datos en una colección en memoria. Observe que la clase `Page` es una clase anidada privada que no forma parte de la interfaz pública. Estos datos se ocultan a los usuarios de esta clase.

Diccionarios

Otro escenario habitual es cuando necesita modelar un diccionario o una asignación. En este escenario, el tipo almacena valores en función de la clave (normalmente claves de texto). En este ejemplo se crea un diccionario que asigna argumentos de la línea de comandos a expresiones lambda que administran estas opciones. En el ejemplo siguiente se muestran dos clases: una clase `ArgsActions`, que asigna una opción de la línea de comandos a un delegado `Action`; y `ArgsProcessor`, que usa `ArgsActions` para ejecutar cada `Action` cuando encuentra esa opción.

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }

}
public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action : defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}
```

En este ejemplo, la colección `ArgsAction` está estrechamente relacionada con la colección subyacente. `get` determina si se ha configurado una opción determinada. Si es así, devuelve la `Action` asociada a esa opción. Si no, devuelve una `Action` que no hace nada. El descriptor de acceso público no incluye ningún descriptor de acceso `set`, sino el diseño que usa un método público para establecer opciones.

Asignaciones multidimensionales

Puede crear indizadores que usen varios argumentos. Además, estos argumentos no se restringen para que sean del mismo tipo. Veamos dos ejemplos.

En el primer ejemplo se muestra una clase que genera valores para el conjunto Mandelbrot. Para obtener más información sobre las matemáticas subyacentes en el conjunto, lea este artículo. El indizador usa dos valores double para definir un punto en el plano X, Y. El descriptor de acceso get calcula el número de iteraciones existente hasta que se determina que un punto no está en el conjunto. Si se alcanza el número máximo de iteraciones, el

punto está en el conjunto y se devuelve el valor maxIterations de la clase (las imágenes generadas por PC popularizadas para el conjunto Mandelbrot definen colores para el número de iteraciones que son necesarias para determinar que un punto en concreto está fuera del conjunto).

```
public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}
```

El conjunto Mandelbrot define valores en cada coordenada (x o y) para los valores numéricos reales. Con esto se define un diccionario que puede contener un número infinito de valores. Por lo tanto, no hay ningún almacenamiento detrás del conjunto. En su lugar, esta clase calcula el valor de cada punto cuando el código llama al descriptor de acceso `get`, por lo que no se usa ningún almacenamiento subyacente.

Vamos a examinar un último uso de los indizadores, en el que el indizador toma varios argumentos de distintos tipos. Imagínese un programa que administra datos históricos de temperaturas. Este indizador usa una ciudad y una fecha para establecer u obtener las temperaturas máximas y mínimas de ese lugar:

```
using DateMeasurements =
    System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>;
using CityDataMeasurements =
    System.Collections.Generic.Dictionary<string, System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

public class HistoricalWeatherData
{
    readonly CityDataMeasurements storage = new CityDataMeasurements();

    public Measurements this[string city, DateTime date]
    {
        get
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
                throw new ArgumentOutOfRangeException(nameof(city), "City not found");

            // strip out any time portion:
            var index = date.Date;
            var measure = default(Measurements);
            if (cityData.TryGetValue(index, out measure))
                return measure;
            throw new ArgumentOutOfRangeException(nameof(date), "Date not found");
        }
        set
        {
            var cityData = default(DateMeasurements);

            if (!storage.TryGetValue(city, out cityData))
            {
                cityData = new DateMeasurements();
                storage.Add(city, cityData);
            }

            // Strip out any time portion:
            var index = date.Date;
            cityData[index] = value;
        }
    }
}
```

Este ejemplo crea un indizador que asigna los datos meteorológicos en dos argumentos diferentes: una ciudad (representada por `string`) y una fecha (representada por `DateTime`). El almacenamiento interno usa dos clases `Dictionary` para representar el diccionario bidimensional. La API pública ya no representa el almacenamiento subyacente. En su lugar, las características del lenguaje de los indizadores le permiten crear una interfaz pública que representa la abstracción, aunque el almacenamiento subyacente debe usar distintos tipos de colección básica.

Hay dos partes de este código que pueden resultar desconocidas para algunos desarrolladores, Estas dos directivas `using`:

```
using DateMeasurements = System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime, IndexersSamples.Common.Measurements>>;
```

Estas instrucciones crean un *alias* para un tipo genérico construido y permiten que el código use después los nombres `DateMeasurements` y `CityDateMeasurements` (más descriptivos) en vez de la construcción genérica de `Dictionary<DateTime, Measurements>` y `Dictionary<string, Dictionary<DateTime, Measurements> >`. Esta construcción requiere el uso de los nombres completos de tipo en el lado derecho del signo `=`.

La segunda técnica consiste en quitar las partes de tiempo de cualquier objeto `DateTime` usado para indizarse en las colecciones. .NET no incluye un tipo de solo fecha. Los desarrolladores usan el tipo `DateTime`, aunque emplean la propiedad `Date` para asegurarse de que cualquier objeto `DateTime` de ese día sea igual.

Resumen

Debe crear indizadores siempre que tenga un elemento de propiedad en la clase, en la que dicha propiedad no representa un valor único, sino una serie de valores donde cada elemento se identifica mediante un conjunto de argumentos. Estos argumentos únicamente pueden identificar el elemento al que se debe hacer referencia en la colección. Los indizadores amplían el concepto de las propiedades, en las que un miembro se trata como un elemento de datos desde fuera de la clase, pero como un método desde dentro. Los indizadores permiten que los argumentos busquen un solo elemento en una propiedad que representa un conjunto de elementos.

Descartes - Guía de C#

18/03/2020 • 12 minutes to read • Edit Online

A partir de C# 7.0, C# admite descartes, que son variables temporales y ficticias que no se usan deliberadamente en el código de la aplicación. Los descartes son equivalentes a variables sin asignar, ya que no tienen un valor. Puesto que solo hay una variable de descarte única y es posible que a dicha variable no se le haya asignado almacenamiento, los descartes pueden reducir las asignaciones de memoria. Como la intención de estas variables es que el código sea claro, mejoran su legibilidad y facilidad de mantenimiento.

Para indicar que una variable es un descarte, se le asigna como nombre el carácter de subrayado (`_`). Por ejemplo, la siguiente llamada al método devuelve una tupla de tres donde el primer y el segundo valor se descartan y *area* es una variable declarada previamente para establecerse en el tercer componente correspondiente devuelto por *GetCityInformation*:

```
(_, _, area) = city.GetCityInformation(cityName);
```

En C# 7.0, se admiten descartes en asignaciones en los contextos siguientes:

- Deconstrucción de tuplas y objetos.
- Coincidencia de patrones con is y switch.
- Llamadas de métodos con parámetros `out`.
- Un carácter de subrayado `_` independiente cuando no hay `_` dentro del ámbito.

Cuando `_` es un descarte válido, si se intenta recuperar su valor o usarlo en una operación de asignación, se genera el error del compilador CS0301: "El nombre '_' no existe en el contexto actual". Esto se debe a que no se le ha asignado un valor a `_`, y es posible que tampoco se le haya asignado una ubicación de almacenamiento. Si fuese una variable real no se podría descartar más de un valor, como en el ejemplo anterior.

Deconstrucción de tuplas y objetos

Los descartes son especialmente útiles en el trabajo con tuplas, cuando el código de la aplicación usa algunos elementos de tupla pero omite otros. Por ejemplo, el siguiente método `QueryCityDataForYears` devuelve una tupla de 6 con el nombre de una ciudad, su superficie, un año, la población de la ciudad en ese año, un segundo año y la población de la ciudad en ese segundo año. En el ejemplo se muestra la evolución de la población entre esos dos años. De los datos disponibles en la tupla, no nos interesa la superficie de la ciudad, y conocemos el nombre de la ciudad y las dos fechas en tiempo de diseño. Como resultado, solo nos interesan los dos valores de población almacenados en la tupla, y podemos controlar los valores restantes como descartes.

```
using System;
using System.Collections.Generic;

public class Example
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");
    }

    private static (string, double, int, int, int, int) QueryCityDataForYears(string name, int year1, int
year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

Para obtener más información sobre la deconstrucción de tuplas con descartes, vea Deconstructing tuples and other types (Deconstruir tuplas y otros tipos).

El método `Deconstruct` de una clase, estructura o interfaz también permite recuperar y deconstruir un conjunto de datos específico de un objeto. Puede usar descartes cuando le interese trabajar con un solo subconjunto de los valores deconstruidos. En el siguiente ejemplo se deconstruye un objeto `Person` en cuatro cadenas (el nombre propio, los apellidos, la ciudad y el estado), pero se descartan los apellidos y el estado.

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class Example
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // <Snippet1>
        // Deconstruct the person object.
        var (fName, _, city, _) = p;
        Console.WriteLine($"Hello {fName} of {city}!");
        // The example displays the following output:
        //      Hello John of Boston!
        // </Snippet1>
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Para obtener más información sobre la deconstrucción de tipos definidos por el usuario con descartes, vea Deconstructing tuples and other types (Deconstruir tuplas y otros tipos).

Coincidencia de patrones con `switch` y `is`

El *patrón de descarte* se puede usar en la coincidencia de patrones con las palabras clave is y switch. Todas las expresiones siempre coinciden con el patrón de descarte.

En el ejemplo siguiente se define un método `ProvidesFormatInfo` que usa instrucciones is para determinar si un objeto proporciona una implementación de IFormatProvider y probar si el objeto es `null`. También se usa el patrón de descarte para controlar los objetos que no son NULL de cualquier otro tipo.

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      object[] objects = { CultureInfo.CurrentCulture,
                           CultureInfo.CurrentCulture.DateTimeFormat,
                           CultureInfo.CurrentCulture.NumberFormat,
                           new ArgumentException(), null };
      foreach (var obj in objects)
         ProvidesFormatInfo(obj);
   }

   private static void ProvidesFormatInfo(object obj)
   {
      switch (obj)
      {
         case IFormatProvider fmt:
            Console.WriteLine($"{fmt} object");
            break;
         case null:
            Console.Write("A null object reference: ");
            Console.WriteLine("Its use could result in a NullReferenceException");
            break;
         case object _:
            Console.WriteLine("Some object type without format information");
            break;
      }
   }
}
// The example displays the following output:
//    en-US object
//    System.Globalization.DateTimeFormatInfo object
//    System.Globalization.NumberFormatInfo object
//    Some object type without format information
//    A null object reference: Its use could result in a NullReferenceException
```

Llamadas de métodos sin parámetros

Cuando se llama al método `Deconstruct` para deconstruir un tipo definido por el usuario (una instancia de una clase, estructura o interfaz), puede descartar los valores de argumentos `out` individuales. Pero también puede descartar el valor de argumentos `out` al llamar a cualquier método con un parámetro out.

En el ejemplo siguiente se llama al método DateTime.TryParse(String, out DateTime) para determinar si la representación de cadena de una fecha es válida en la referencia cultural actual. Dado que al ejemplo solo le interesa validar la cadena de fecha, y no analizarla para extraer la fecha, el argumento `out` para el método es un descarte.

```
using System;

public class Example
{
   public static void Main()
   {
      string[] dateStrings = {"05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                              "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                              "5/01/2018 14:57:32.80 -07:00",
                              "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                              "Fri, 15 May 2018 20:10:57 GMT" };
      foreach (string dateString in dateStrings)
      {
         if (DateTime.TryParse(dateString, out _))
            Console.WriteLine($"'{dateString}': valid");
         else
            Console.WriteLine($"'{dateString}': invalid");
      }
   }
}
// The example displays output like the following:
//       '05/01/2018 14:57:32.8': valid
//       '2018-05-01 14:57:32.8': valid
//       '2018-05-01T14:57:32.8375298-04:00': valid
//       '5/01/2018': valid
//       '5/01/2018 14:57:32.80 -07:00': valid
//       '1 May 2018 2:57:32.8 PM': valid
//       '16-05-2018 1:00:32 PM': invalid
//       'Fri, 15 May 2018 20:10:57 GMT': invalid
```

Descarte independiente

Puede usar un descarte independiente para indicar cualquier variable que decida omitir. En el ejemplo siguiente se usa un descarte independiente para omitir el objeto Task devuelto por una operación asincrónica. Como resultado, se suprime la excepción que se produce en la operación cuando está a punto de completarse.

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static async Task Main(string[] args)
   {
      await ExecuteAsyncMethods();
   }

   private static async Task ExecuteAsyncMethods()
   {
      Console.WriteLine("About to launch a task...");
      _ = Task.Run(() => { var iterations = 0;
                           for (int ctr = 0; ctr < int.MaxValue; ctr++)
                              iterations++;
                           Console.WriteLine("Completed looping operation...");
                           throw new InvalidOperationException();
                         });
      await Task.Delay(5000);
      Console.WriteLine("Exiting after 5 second delay");
   }
}
// The example displays output like the following:
//       About to launch a task...
//       Completed looping operation...
//       Exiting after 5 second delay
```

Tenga en cuenta que `_` también es un identificador válido. Cuando se usa fuera de un contexto compatible, `_` no se trata como un descarte, sino como una variable válida. Si un identificador denominado `_` ya está en el ámbito, el uso de `_` como descarte independiente puede producir lo siguiente:

- La modificación accidental del valor de la variable `_` en el ámbito, al asignarle el valor del descarte previsto. Por ejemplo:

```
private static void ShowValue(int _)
{
   byte[] arr = { 0, 0, 1, 2 };
   _ = BitConverter.ToInt32(arr, 0);
   Console.WriteLine(_);
}
// The example displays the following output:
//       33619968
```

- Un error del compilador por infringir la seguridad de tipos. Por ejemplo:

```
private static bool RoundTrips(int _)
{
   string value = _.ToString();
   int newValue = 0;
   _ = Int32.TryParse(value, out newValue);
   return _ == newValue;
}
// The example displays the following compiler error:
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'
```

- Error del compilador CS0136: "Una variable local o un parámetro denominados '_' no se pueden declarar en este ámbito porque ese nombre se está usando en un ámbito local envolvente para definir una variable local o un parámetro". Por ejemplo:

```
public void DoSomething(int _)
{
 var _ = GetValue(); // Error: cannot declare local _ when one is already in scope
}
// The example displays the following compiler error:
// error CS0136:
//       A local or parameter named '_' cannot be declared in this scope
//       because that name is used in an enclosing local scope
//       to define a local or parameter
```

Vea también

- Deconstrucción de tuplas y otros tipos
- Palabra clave `is`
- Palabra clave `switch`

Genéricos (Guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

Los genéricos introducen el concepto de parámetros de tipo a .NET, lo que le permite diseñar clases y métodos que aplazan la especificación de uno o varios tipos hasta que el código de cliente declare y cree una instancia de la clase o el método. Por ejemplo, al usar un parámetro de tipo genérico `T`, puede escribir una clase única que otro código de cliente puede usar sin incurrir en el costo o riesgo de conversiones en tiempo de ejecución u operaciones de conversión boxing, como se muestra aquí:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Las clases y métodos genéricos combinan reusabilidad, seguridad de tipos y eficacia de una manera en que sus homólogos no genéricos no pueden. Los genéricos se usan frecuentemente con colecciones y los métodos que funcionan en ellas. El espacio de nombres System.Collections.Generic contiene varias clases de colecciones basadas en genéricos. No se recomiendan las colecciones no genéricas, como ArrayList, y se mantienen por compatibilidad. Para más información, vea Elementos genéricos en .NET.

Por supuesto, también se pueden crear tipos y métodos genéricos personalizados para proporcionar soluciones y patrones de diseño generalizados propios con seguridad de tipos y eficaces. En el ejemplo de código siguiente se muestra una clase genérica simple de lista vinculada para fines de demostración. (En la mayoría de los casos, debe usar la clase List<T> proporcionada por .NET en lugar de crear la suya propia). El parámetro de tipo `T` se usa en diversas ubicaciones donde normalmente se usaría un tipo concreto para indicar el tipo del elemento almacenado en la lista. Se usa de estas formas:

- Como el tipo de un parámetro de método en el método `AddHead`.
- Como el tipo de valor devuelto de la propiedad `Data` en la clase anidada `Node`.
- Como el tipo de miembro privado `data` de la clase anidada.

Tenga en cuenta que `T` está disponible para la clase anidada `Node`. Cuando se crea una instancia de `GenericList<T>` con un tipo concreto, por ejemplo como un `GenericList<int>`, cada repetición de `T` se sustituye por `int`.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

En el ejemplo de código siguiente se muestra cómo el código de cliente usa la clase genérica `GenericList<T>` para crear una lista de enteros. Simplemente cambiando el argumento de tipo, el código siguiente puede modificarse fácilmente para crear listas de cadenas o cualquier otro tipo personalizado:

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Introducción a los genéricos

- Use tipos genéricos para maximizar la reutilización del código, la seguridad de tipos y el rendimiento.
- El uso más común de los genéricos es crear clases de colección.
- La biblioteca de clases de .NET contiene varias clases de colección genéricas en el espacio de nombres System.Collections.Generic. Estas se deberían usar siempre que sea posible en lugar de clases como ArrayList en el espacio de nombres System.Collections.
- Puede crear sus propias interfaces, clases, métodos, eventos y delegados genéricos.
- Puede limitar las clases genéricas para habilitar el acceso a métodos en tipos de datos determinados.
- Puede obtener información sobre los tipos que se usan en un tipo de datos genérico en tiempo de ejecución mediante la reflexión.

Secciones relacionadas

- Parámetros de tipo genérico
- Restricciones de tipos de parámetros
- Clases genéricas
- Interfaces genéricas
- Métodos genéricos
- Delegados genéricos
- Diferencias entre plantillas de C++ y tipos genéricos de C#
- Genéricos y reflexión
- Genéricos en el motor en tiempo de ejecución

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#.

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Tipos
- <typeparam>

- <typeparamref>
- Elementos genéricos en .NET

Iterators

06/05/2020 • 10 minutes to read • Edit Online

Prácticamente todos los programas que escriba tendrán alguna necesidad de recorrer en iteración una colección. Va a escribir código que examine cada elemento de una colección.

También va a crear métodos de iterador, que son los métodos que genera un iterador (un objeto que atraviesa un contenedor, especialmente listas) para los elementos de esa clase. Se pueden usar para:

- Realizar una acción en cada elemento de una colección.
- Enumerar una colección personalizada.
- Extender LINQ u otras bibliotecas.
- Crear una canalización de datos en la que los datos fluyan de forma eficaz mediante métodos de iterador.

El lenguaje C# proporciona características para ambos escenarios. Este artículo proporciona información general sobre esas características.

Este tutorial consta de varios pasos. Después de cada paso, puede ejecutar la aplicación y ver el progreso. También puede ver o descargar el ejemplo completo para este tema. Para obtener instrucciones de descarga, vea Ejemplos y tutoriales.

Iteración con foreach

Enumerar una colección es sencillo: la palabra clave `foreach` enumera una colección, ejecutando la instrucción incrustada una vez para cada elemento de la colección:

```
foreach (var item in collection)
{
   Console.WriteLine(item.ToString());
}
```

Así de simple. Para recorrer en iteración todo el contenido de una colección, la instrucción `foreach` es todo lo que necesita. Pero la instrucción `foreach` no es mágica. Depende de dos interfaces genéricas definidas en la biblioteca de .NET Core para generar el código necesario para recorrer en iteración una colección: `IEnumerable<T>` e `IEnumerator<T>`. Este mecanismo se explica con más detalle a continuación.

Ambas interfaces tienen también homólogas no genéricas: `IEnumerable` e `IEnumerator`. Para el código moderno se prefieren las versiones genéricas.

Orígenes de enumeración con métodos de iterador

Otra magnífica característica del lenguaje C# permite generar métodos que crean un origen para una enumeración. Se conocen como *métodos de iterador*. Un método de iterador define cómo generar los objetos de una secuencia cuando se solicita. Para definir un método de iterador se usan las palabras clave contextuales `yield return`.

Podría escribir este método para generar la secuencia de enteros de 0 a 9:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

El código anterior muestra instrucciones distintivas `yield return` para resaltar el hecho de que se pueden usar varias instrucciones discretas `yield return` en un método de iterador. Puede usar (y hágalo a menudo) otras construcciones de lenguaje para simplificar el código de un método de iterador. La definición del método siguiente genera la misma secuencia de números:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}
```

No tiene que elegir entre una y otra. Puede tener tantas instrucciones `yield return` como sea necesario para satisfacer las necesidades del método:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

Esa es la sintaxis básica. Veamos un ejemplo del mundo real en el que se escribe un método de iterador. Imagine que se encuentra en un proyecto de IoT y los sensores del dispositivo generan un flujo de datos muy grande. Para hacerse una idea de los datos, podría escribir un método que tomara muestras de cada enésimo elemento de datos. Este pequeño método de iterador lo hace:

```
public static IEnumerable<T> Sample(this IEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

Hay una restricción importante en los métodos de iterador: no puede tener una instrucción `return` y una

instrucción `yield return` en el mismo método. Lo siguiente no se compilará:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    return items;
}
```

Normalmente esta restricción no supone un problema. Tiene la opción de usar `yield return` en todo el método o de separar el método original en varios métodos, unos con `return` y otros con `yield return`.

Puede modificar el último método ligeramente para usar `yield return` en todas partes:

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109 };
    foreach (var item in items)
        yield return item;
}
```

A veces, la respuesta correcta es dividir un método de iterador en dos métodos distintos. Uno que use `return` y un segundo que use `yield return`. Imagine una situación en la que quiera devolver una colección vacía, o los 5 primeros números impares, basándose en un argumento booleano. Eso se podría escribir como estos dos métodos:

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}
```

Observe los métodos anteriores. El primero usa la instrucción estándar `return` para devolver una colección vacía o el iterador creado por el segundo método. El segundo método usa la instrucción `yield return` para crear la secuencia solicitada.

Profundización en `foreach`

La instrucción `foreach` se expande en un elemento estándar que usa las interfaces `IEnumerable<T>` e `IEnumerator<T>` para recorrer en iteración todos los elementos de una colección. También minimiza los errores cometidos por los desarrolladores al no administrar correctamente los recursos.

El compilador traduce el bucle `foreach` que se muestra en el primer ejemplo en algo similar a esta construcción:

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

La construcción anterior representa el código generado por el compilador de C# a partir de la versión 5 y superiores. Antes de la versión 5, la variable `item` tenía un ámbito diferente:

```
// C# versions 1 through 4:
IEnumerator<int> enumerator = collection.GetEnumerator();
int item = default(int);
while (enumerator.MoveNext())
{
    item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

Se modificó porque el comportamiento anterior podía provocar errores leves y difíciles de diagnosticar en las expresiones lambda. Para más información sobre las expresiones lambda, consulte Expresiones lambda.

El código exacto generado por el compilador es algo más complicado y controla las situaciones en las que el objeto devuelto por `GetEnumerator()` implementa la interfaz `IDisposable`. La expansión completa genera código más parecido al siguiente:

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    } finally
    {
        // dispose of enumerator.
    }
}
```

La manera en que el enumerador se elimina depende de las características del tipo de `enumerator`. En el caso general, la cláusula `finally` se expande en:

```
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

Pero si el tipo de `enumerator` es un tipo sellado y no hay conversión implícita del tipo de `enumerator` a `IDisposable`, la cláusula `finally` se expande en un bloque vacío:

```
finally
{
}
```

Si hay una conversión implícita del tipo de `enumerator` a `IDisposable`, y `enumerator` es un tipo de valor que no acepta valores Null, la cláusula `finally` se expande en:

```
finally
{
   ((IDisposable)enumerator).Dispose();
}
```

Afortunadamente, no es necesario recordar todos estos detalles. La instrucción `foreach` controla todos esos matices. El compilador generará el código correcto para cualquiera de estas construcciones.

Introducción a los delegados

18/03/2020 • 4 minutes to read • Edit Online

Los delegados proporcionan un mecanismo de *enlace en tiempo de ejecución* en .NET. Un enlace en tiempo de ejecución significa que se crea un algoritmo en el que el llamador también proporciona al menos un método que implementa parte del algoritmo.

Por ejemplo, considere la ordenación de una lista de estrellas en una aplicación de astronomía. Puede decidir ordenar las estrellas por su distancia con respecto a la Tierra, por la magnitud de la estrella o por su brillo percibido.

En todos estos casos, el método Sort() hace básicamente lo mismo: organiza los elementos en la lista en función de una comparación. El código que compara dos estrellas es diferente para cada uno de los criterios de ordenación.

Este tipo de soluciones se ha usado en software durante aproximadamente medio siglo. El concepto de delegado del lenguaje C# proporciona compatibilidad con el lenguaje de primera clase y seguridad de tipos en torno a este concepto.

Como verá más adelante en esta serie, el código de C# que escriba para algoritmos como este posee seguridad de tipos y aprovecha el lenguaje y el compilador para asegurarse de que los tipos coincidan con los argumentos y los tipos devueltos.

Objetivos del diseño de lenguaje para los delegados

Los diseñadores de lenguaje enumeraron varios objetivos para la característica que se acabó convirtiendo en los delegados.

El equipo aspiraba a crear una construcción de lenguaje común que pudiera usarse para cualquier algoritmo de enlace en tiempo de ejecución. Esto permite a los desarrolladores aprender un concepto y usarlo en muchos problemas de software diferentes.

En segundo lugar, el equipo quería que se admitiesen llamadas a métodos únicos y multidifusión. Los delegados de multidifusión son delegados que encadenan varias llamadas a métodos. Se verán ejemplos más adelante en esta serie.

El equipo quería que los delegados admitiesen la misma seguridad de tipos que los desarrolladores esperan de todas las construcciones C#.

Por último, el equipo reconocía que un patrón de eventos es un patrón específico en el que los delegados, o cualquier algoritmo de enlace en tiempo de ejecución, resultan muy útiles. El equipo quería garantizar que el código de los delegados proporcionase una base para el patrón de eventos de .NET.

El resultado de todo ese trabajo fue la compatibilidad con los delegados y los eventos en C# y .NET. Los artículos restantes de esta sección tratarán sobre las características del lenguaje, la compatibilidad con bibliotecas y las expresiones comunes que se usan al trabajar con los delegados.

Obtendrá información sobre la palabra clave `delegate` y el código que genera. Conocerá las características de la clase `System.Delegate` y la manera en que se usan dichas características. Aprenderá a crear delegados con seguridad de tipos y métodos que se pueden invocar a través de delegados. También aprenderá a trabajar con delegados y eventos mediante expresiones lambda. Verá como los delegados se convierten en uno de los bloques de creación para LINQ. Descubrirá que los delegados son la base del patrón de eventos de .NET y entenderá en qué se diferencian.

En resumen, constatará que los delegados son una parte integral de la programación en .NET y de las operaciones con API de marco de trabajo.

Comencemos.

Siguiente

System.Delegate y la palabra clave `delegate`

18/03/2020 • 13 minutes to read • Edit Online

Anterior

En este artículo se tratan las clases de .NET que admiten delegados y sobre cómo se asignan a la palabra clave `delegate`.

Definición de los tipos delegados

Comencemos con la palabra clave "delegate", ya que es lo que usará principalmente al trabajar con delegados. El código que genere el compilador cuando se usa la palabra clave `delegate` se asignará a las llamadas de método que invocan a miembros de las clases Delegate y MulticastDelegate.

Para definir un tipo de delegado, se usa una sintaxis similar a la definición de una firma de método. Solo hace falta agregar la palabra clave `delegate` a la definición.

Vamos a usar el método List.Sort() como ejemplo. El primer paso consiste en crear un tipo para el delegado de comparación:

```
// From the .NET Core library

// Define the delegate type:
public delegate int Comparison<in T>(T left, T right);
```

El compilador genera una clase derivada de `System.Delegate` que coincide con la firma usada (en este caso, un método que devuelve un entero y tiene dos argumentos). El tipo de ese delegado es `Comparison`. El tipo de delegado `Comparison` es un tipo genérico. Aquí puede obtener más información sobre los genéricos.

Observe que puede parecer que la sintaxis declara una variable, pero en realidad declara un *tipo*. Puede definir tipos de delegado dentro de clases, directamente dentro de espacios de nombres o incluso en el espacio de nombres global.

> **NOTE**
> No se recomienda declarar tipos de delegado (u otros tipos) directamente en el espacio de nombres global.

El compilador también genera controladores de adición y eliminación para este nuevo tipo, de modo que los clientes de esta clase puedan agregar y quitar métodos de la lista de invocación de una instancia. El compilador forzará que la firma del método que se agrega o se quita coincida con la firma usada al declarar el método.

Declaración de instancias de delegados

Después de definir el delegado, puede crear una instancia de ese tipo. Al igual que todas las variables en C#, no puede declarar instancias de delegados directamente en un espacio de nombres o en el espacio de nombres global.

```
// inside a class definition:

// Declare an instance of that type:
public Comparison<T> comparator;
```

El tipo de la variable es `Comparison<T>`, el tipo de delegado definido anteriormente. El nombre de la variable es `comparator`.

El fragmento de código anterior declara una variable de miembro dentro de una clase. También puede declarar variables de delegado que sean variables locales o argumentos para los métodos.

Invocación de delegados

Para invocar los métodos que se encuentran en la lista de invocación de un delegado, llame a dicho delegado. Dentro del método `Sort()`, el código llamará al método de comparación para determinar en qué orden colocará los objetos:

```
int result = comparator(left, right);
```

En la línea anterior, el código *invoca* al método asociado al delegado. La variable se trata como un nombre de método y se invoca mediante la sintaxis de llamada de método normal.

Esta línea de código realiza una suposición arriesgada, ya que no hay ninguna garantía de que se haya agregado un destino al delegado. Si no se ha asociado ningún destino, la línea anterior haría que se produjese una `NullReferenceException`. Las expresiones que se usan para resolver este problema son más complicadas que una simple comprobación de null y se tratan más adelante en esta serie.

Asignación, adición y eliminación de destinos de invocación

Hemos visto cómo se define un tipo de delegado y cómo se declaran y se invocan las instancias de delegado.

Los programadores que quieran usar el método `List.Sort()` deben definir un método cuya firma coincida con la definición del tipo de delegado y asignarlo al delegado usado por el método de ordenación. Esta asignación agrega el método a la lista de invocación de ese objeto de delegado.

Supongamos que quiera ordenar una lista de cadenas por su duración. La función de comparación podría ser la siguiente:

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

El método se ha declarado como un método privado. Esto es correcto, ya que tal vez no le interese que este método forme parte de la interfaz pública. Aun así, puede usarse como método de comparación cuando se asocia a un delegado. El código de llamada tendrá este método asociado a la lista de destino del objeto de delegado y puede tener acceso a él a través de ese delegado.

Para crear esta relación, pase ese método al método `List.Sort()`:

```
phrases.Sort(CompareLength);
```

Observe que se usa el nombre del método sin paréntesis. Al usar el método como un argumento, le indica al compilador que convierta la referencia del método en una referencia que se puede usar como un destino de invocación del delegado y que asocie ese método como un destino de invocación.

También podría haber declarado de forma explícita una variable de tipo `Comparison<string>` y realizado una asignación:

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

En los casos en los que el método que se usa como destino del delegado es un método pequeño, es habitual usar la sintaxis de expresión lambda para realizar la asignación:

```
Comparison<string> comparer = (left, right) => left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

El uso de expresiones lambda para destinos de delegados se explica con más detalle en una sección posterior.

En el ejemplo de Sort() se suele asociar un método de destino único al delegado, pero los objetos delegados admiten listas de invocación que tienen varios métodos de destino asociados a un objeto delegado.

Clases Delegate y MulticastDelegate

La compatibilidad de lenguaje descrita anteriormente proporciona las características y la compatibilidad que normalmente necesitará para trabajar con delegados. Estas características están integradas en dos clases en .NET Core Framework: Delegate y MulticastDelegate.

La clase `System.Delegate` y su única subclase directa `System.MulticastDelegate` proporcionan la compatibilidad con el marco para crear delegados, registrar métodos como destinos de delegados e invocar todos los métodos que se registran como un destino del delegado.

Curiosamente, las clases `System.Delegate` y `System.MulticastDelegate` no son tipos de delegado, pero proporcionan la base para todos los tipos de delegado específicos. El propio proceso de diseño del lenguaje dictaba que no se puede declarar una clase que derive de `Delegate` o `MulticastDelegate`. Las reglas del lenguaje C# lo prohíben.

En cambio, el compilador de C# crea instancias de una clase derivada de `MulticastDelegate` cuando se usa la palabra clave del lenguaje C# para declarar tipos de delegado.

Este diseño tiene sus orígenes en la primera versión de C# y. NET. Uno de los objetivos del equipo de diseño era asegurarse de que el lenguaje aplicaba seguridad de tipos al usar delegados. Esto significaba que debían asegurarse no solo de que los delegados se invocaban con el tipo y el número adecuados de argumentos, sino de que todos los tipos de valor devueltos se indicaban correctamente en tiempo de compilación. Los delegados formaban parte de la versión 1.0 .NET, que apareció antes que los genéricos.

La mejor manera de aplicar la seguridad de tipos era que el compilador crease las clases de delegado concretas que representasen la firma del método que se usaba.

Aunque usted no puede crear clases derivadas directamente, usará los métodos definidos en estas clases. Vamos a ver los métodos más comunes que usará al trabajar con delegados.

Lo primero que debe recordar es que cada delegado con el que trabaje se deriva de `MulticastDelegate`. Un delegado multidifusión significa que se puede invocar más de un destino de método al invocar en un delegado. El diseño original consideraba la posibilidad de establecer una distinción entre los delegados en los que solo se podía asociar e invocar un método de destino y los delegados en los que se podía asociar e invocar varios métodos de destino. Esa distinción resultó ser menos útil en la práctica de lo que se pensaba. Las dos clases ya estaban creadas y se han conservado en el marco de trabajo desde la versión pública inicial.

Los métodos que usará más a menudo con los delegados son `Invoke()` y `BeginInvoke()` / `EndInvoke()`.

`Invoke()` invocará todos los métodos que se han asociado a una instancia de delegado en concreto. Como ya hemos visto anteriormente, por lo general los delegados se invocan mediante la sintaxis de llamada de método en la variable de delegado. Como verá más adelante en esta serie, hay patrones que funcionan directamente con estos métodos.

Ahora que ha visto la sintaxis del lenguaje y las clases que admiten delegados, examinemos cómo se usan, se crean y se invocan delegados fuertemente tipados.

Siguiente

Delegados fuertemente tipados

18/03/2020 • 5 minutes to read • Edit Online

Anterior

En el artículo anterior pudo ver que con la palabra clave `delegate` se crean tipos de delegados concretos.

La clase abstracta Delegate proporciona la infraestructura para el acoplamiento débil y la invocación. Los tipos de delegado concretos se hacen mucho más útiles al adoptar y aplicar la seguridad de tipos para los métodos agregados a la lista de invocación de un objeto de delegado. Cuando se usa palabra clave `delegate` y se define un tipo de delegado concreto, el compilador genera esos métodos.

En la práctica, esto daría lugar a la creación de nuevos tipos de delegado siempre que necesitara otra firma de método. Este trabajo podría resultar tedioso pasado un tiempo. Cada nueva característica exige nuevos tipos de delegado.

Afortunadamente, esto no es necesario. .NET Core Framework contiene varios tipos que puede volver a usar siempre que necesite tipos de delegado. Son definiciones genéricas, por lo que puede declarar personalizaciones cuando necesite nuevas declaraciones de método.

El primero de estos tipos es el tipo Action y distintas variaciones:

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

El modificador `in` del argumento de tipo genérico se trata en el artículo sobre la covarianza.

Hay variaciones del delegado `Action` que contienen hasta 16 argumentos, como Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>. Es importante que estas definiciones usen argumentos genéricos distintos para cada uno de los argumentos del delegado: eso proporciona la máxima flexibilidad. Los argumentos de método no tienen que ser, aunque pueden ser, del mismo tipo.

Use uno de los tipos `Action` para cualquier tipo de delegado que tenga un tipo de valor devuelto void.

.NET Framework también incluye varios tipos de delegado genéricos que se pueden usar para los tipos de delegado que devuelven valores:

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

El modificador `out` del argumento de tipo genérico result se trata en el artículo sobre la covarianza.

Hay variaciones del delegado `Func` con hasta 16 argumentos de entrada, como Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>. Por convención, el tipo del resultado siempre es el último parámetro de tipo de todas las declaraciones `Func`.

Use uno de los tipos `Func` para cualquier tipo de delegado que devuelva un valor.

También hay un tipo Predicate<T> especializado para un delegado que devuelva una prueba sobre un valor único:

```
public delegate bool Predicate<in T>(T obj);
```

Es posible que observe que para cualquier tipo `Predicate` existe un tipo `Func` estructuralmente equivalente, por ejemplo:

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

Podría llegar a pensar que estos dos tipos son equivalentes, pero no lo son. Estas dos variables no se pueden usar indistintamente. A una variable de un tipo no se le puede asignar el otro tipo. El sistema de tipos de C# usa los nombres de los tipos definidos, no la estructura.

Todas estas definiciones de tipos de delegado de la biblioteca de .NET Core deberían significar que no es necesario definir ningún tipo de delegado nuevo para cualquier característica nueva creada que exija delegados. Estas definiciones genéricas deberían proporcionar todos los tipos de delegado necesarios para la mayoría de las situaciones. Puede simplemente crear instancias de uno de estos tipos con los parámetros de tipo necesarios. En el caso de los algoritmos que se pueden convertir en genéricos, estos delegados se pueden usar como tipos genéricos.

Esto debería ahorrar tiempo y minimizar el número de nuevos tipos que es necesario crear para poder trabajar con delegados.

En el siguiente artículo se verán varios patrones comunes para trabajar con delegados en la práctica.

Siguiente

Patrones comunes para delegados

18/03/2020 • 15 minutes to read • Edit Online

Anterior

Los delegados proporcionan un mecanismo que permite que los diseños de software supongan un acoplamiento mínimo entre los componentes.

Un ejemplo excelente de este tipo de diseño es LINQ. El patrón de expresión de consulta LINQ se basa en los delegados para todas sus características. Considere este ejemplo sencillo:

```
var smallNumbers = numbers.Where(n => n < 10);
```

Se filtra la secuencia solo de los números que son inferiores al valor 10. El método `Where` usa un delegado que determina qué elementos de una secuencia pasan el filtro. Cuando crea una consulta LINQ, proporciona la implementación del delegado para este fin específico.

El prototipo para el método Where es:

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool>
predicate);
```

Este ejemplo se repite con todos los métodos que forman parte de LINQ. Todos se basan en delegados para el código que administra la consulta específica. Este modelo de diseño de API es muy eficaz para obtener información y comprender.

En este ejemplo sencillo se ilustra cómo los delegados necesitan muy poco acoplamiento entre componentes. No necesita crear una clase que derive de una clase base determinada. No necesita implementar una interfaz específica. El único requisito consiste en proporcionar la implementación de un método que sea fundamental para la tarea que nos ocupa.

Crear sus propios componentes con delegados

Vamos a aprovechar ese ejemplo creando un componente con un diseño que se base en delegados.

Vamos a definir un componente que pueda usarse para mensajes de registro en un sistema grande. Los componentes de la biblioteca pueden usarse en muchos entornos diferentes, en varias plataformas diferentes. Existen muchas características comunes en el componente que administra los registros. Necesitará aceptar mensajes de cualquier componente del sistema. Esos mensajes tendrán diferentes prioridades, que el componente principal puede administrar. Los mensajes deben tener marcas de tiempo en su forma de archivado final. Para escenarios más avanzados, puede filtrar mensajes por el componente de origen.

Hay un aspecto de la característica que cambiará a menudo: el lugar donde se escriben los mensajes. En algunos entornos, pueden escribirse en la consola de errores. En otros, en un archivo. Otras posibilidades incluyen el almacenamiento de bases de datos, los registros de eventos del sistema operativo u otro almacenamiento de documentos.

También hay combinaciones de salida que pueden usarse en escenarios diferentes. Puede que quiera escribir mensajes en la consola y en un archivo.

Un diseño basado en delegados proporcionará una gran flexibilidad y facilitará la compatibilidad de mecanismos de almacenamiento que pueden agregarse en el futuro.

Con este diseño, el componente de registro principal puede ser una clase no virtual, incluso sellada. Puede conectar cualquier conjunto de delegados para escribir los mensajes en un medio de almacenamiento diferente. La compatibilidad integrada para los delegados multidifusión facilita la compatibilidad de escenarios donde los mensajes deben escribirse en varias ubicaciones (un archivo y una consola).

Una primera implementación

Comencemos poco a poco: la implementación inicial aceptará mensajes nuevos y los escribirá con cualquier delegado asociado. Puede comenzar con un delegado que escriba mensajes en la consola.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        WriteMessage(msg);
    }
}
```

La clase estática anterior es lo más sencillo que puede funcionar. Necesitamos escribir solo la implementación para el método que escribe mensajes en la consola:

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Por último, necesita conectar el delegado asociándolo al delegado WriteMessage que se declara en el registrador:

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Procedimientos

Hasta ahora nuestro ejemplo es bastante sencillo, pero sigue mostrando algunas instrucciones importantes para los diseños que involucran a los delegados.

Con los tipos de delegado definidos en Core Framework es más sencillo para los usuarios trabajar con los delegados. No necesita definir tipos nuevos, y los desarrolladores que usen su biblioteca no necesitan aprender nuevos tipos de delegado especializados.

Las interfaces que se han usado son tan mínimas y flexibles como es posible: para crear un registrador de salida nuevo, debe crear un método. Ese método puede ser un método estático o un método de instancia. Puede tener cualquier acceso.

Resultados con formato

Vamos a hacer esta primera versión un poco más sólida y, después, empezaremos a crear otros mecanismos de registro.

Después, vamos a agregar algunos argumentos al método `LogMessage()` de manera que su clase de registro cree más mensajes estructurados:

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

A continuación, vamos a usar ese argumento `Severity` para filtrar los mensajes que se envían a la salida del registro.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static Severity LogLevel {get;set;} = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        WriteMessage(outputMsg);
    }
}
```

Procedimientos

Ha agregado características nuevas a la infraestructura de registro. Como el componente del registrador se acopla débilmente a cualquier mecanismo de salida, estas características nuevas pueden agregarse sin afectar a ningún código que implementa el delegado del registrador.

A medida que siga creando esto, verá más ejemplos de cómo este acoplamiento débil permite una mayor flexibilidad en la actualización de las partes del sitio sin que haya cambios en otras ubicaciones. De hecho, en una aplicación más grande, las clases de salida del registrador pueden estar en un ensamblado diferente, y ni siquiera necesitan volver a crearse.

Crear un segundo motor de salida

El componente de registro se está desarrollando correctamente. Vamos a agregar un motor de salida más que registre mensajes en un archivo. Este será un motor de salida ligeramente más involucrado. Será una clase que encapsule las operaciones de archivo y garantice que el archivo esté siempre cerrado después de cada escritura. Eso garantiza que todos los datos se vacíen en el disco después de que se genere cada mensaje.

Aquí se muestra ese registrador basado en archivos:

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

Una vez que haya creado esta clase, puede inicializarla y esta asocia su método LogMessage al componente de registrador:

```
var file = new FileLogger("log.txt");
```

Estos dos no son mutuamente exclusivos. Puede asociar ambos métodos de registro y generar mensajes en la consola y en un archivo:

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the static class we utilized
earlier
```

Después, incluso en la misma aplicación, puede quitar uno de los delegados sin ocasionar ningún otro problema en el sistema:

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

Procedimientos

Ahora, ha agregado un segundo controlador de salida para el subsistema de registro. Este necesita un poco más de infraestructura para admitir correctamente el sistema de archivos. El delegado es un método de instancia. También es un método privado. No existe ninguna necesidad de una mayor accesibilidad porque la infraestructura de delegado puede conectarse a los delegados.

En segundo lugar, el diseño basado en delegados permite varios métodos de salida sin ningún código adicional. No necesita crear ninguna infraestructura adicional para admitir varios métodos de salida. Simplemente se convierten en otro método en la lista de invocación.

Preste una atención especial al código del método de salida de registro de archivo. Se codifica para garantizar que no produce ninguna excepción. Aunque esto no siempre es estrictamente necesario, a menudo es un buen procedimiento. Si cualquiera de los métodos de delegado produce una excepción, los delegados restantes que se encuentran en la invocación no se invocarán.

Como última observación, el registrador de archivos debe administrar sus recursos abriendo y cerrando el archivo en cada mensaje de registro. Puede optar por mantener el archivo abierto e implementar IDisposable para cerrar el archivo cuando termine. Cualquier método tiene sus ventajas e inconvenientes. Ambos crean un poco más de acoplamiento entre las clases.

Ningún código de la clase de registrador necesitará actualizarse para admitir cualquier escenario.

Control de delegados NULL

Por último, vamos a actualizar el método LogMessage de manera que sea sólido para esos casos en los que no se selecciona ningún mecanismo de salida. La implementación actual producirá `NullReferenceException` cuando el delegado `WriteMessage` no tenga una lista de invocación asociada. Puede que prefiera un diseño que continúe silenciosamente cuando no se haya asociado ningún método. Esto es sencillo con el operador condicional NULL, combinado con el método `Delegate.Invoke()`:

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

El operador condicional NULL (`?.`) crea un cortocircuito cuando el operando izquierdo (`WriteMessage` en este caso) es NULL, lo que significa que no se realiza ningún intento para registrar un mensaje.

No encontrará el método `Invoke()` en la documentación de `System.Delegate` o `System.MulticastDelegate`. El compilador genera un método `Invoke` con seguridad de tipos para cualquier tipo de delegado declarado. En este ejemplo, eso significa que `Invoke` toma un solo argumento `string` y tiene un tipo de valor devuelto void.

Resumen de procedimientos

Ha observado los comienzos de un componente de registro que puede expandirse con otros sistemas de escritura y otras características. Con el uso de delegados en el diseño, estos componentes diferentes están acoplados muy débilmente. Esto ofrece varias ventajas. Es muy sencillo crear mecanismos de salida nuevos y asociarlos al sistema. Estos otros mecanismos solo necesitan un método: el método que escribe el mensaje de registro. Es un diseño que es muy resistente cuando se agregan características nuevas. El contrato que se necesita para cualquier sistema de escritura es implementar un método. Ese método puede ser un método estático o de instancia. Puede ser público, privado o de cualquier otro acceso legal.

La clase de registrador puede realizar cualquier número de cambios o mejoras sin producir cambios importantes. Como cualquier clase, no puede modificar la API pública sin el riesgo de que se produzcan cambios importantes. Pero, como el acoplamiento entre el registrador y cualquier motor de salida se realiza solo mediante el delegado, ningún otro tipo (como interfaces o clases base) está involucrado. El acoplamiento es lo más pequeño posible.

Siguiente

Introducción a los eventos

18/03/2020 • 6 minutes to read • Edit Online

Anterior

Los eventos son, como los delegados, un mecanismo de *enlace en tiempo de ejecución*. De hecho, los eventos se crean con compatibilidad de lenguaje para los delegados.

Los eventos son una manera para que un objeto difunda (a todos los componentes interesados del sistema) que algo ha sucedido. Cualquier otro componente puede suscribirse al evento, y recibir una notificación cuando se genere uno.

Probablemente ha usado eventos en alguna programación. Muchos sistemas gráficos tienen un modelo de eventos para notificar la interacción del usuario. Estos eventos notificarán movimiento del mouse, pulsaciones de botón e interacciones similares. Ese es uno de los más comunes, pero realmente no es el único escenario donde se usan eventos.

Puede definir eventos que deben generarse para las clases. Una consideración importante a la hora de trabajar con eventos es que puede que no haya ningún objeto registrado para un evento determinado. Debe escribir el código de manera que no genere eventos cuando no esté configurado ningún agente de escucha.

La suscripción a un evento también crea un acoplamiento entre dos objetos (el origen del evento y el receptor del evento). Necesita asegurarse de que el receptor del evento cancela la suscripción del origen del evento cuando ya no está interesado en eventos.

Diseño de objetivos para la compatibilidad con eventos

El diseño del lenguaje para eventos tiene como destino estos objetivos:

- Permita un acoplamiento mínimo entre un origen de eventos y un receptor de eventos. Estos dos componentes puede que no se hayan escrito por la misma organización e incluso pueden actualizarse en programaciones totalmente diferentes.
- Debe ser muy sencillo suscribirse a un evento y cancelar la suscripción de este.
- Los orígenes de eventos deben admitir varios suscriptores de eventos. También deben admitir no tener ningún suscriptor de eventos asociado.

Puede observar que los objetivos de los eventos son muy similares a los de los delegados. Este es el motivo por el que la compatibilidad del lenguaje de eventos se basa en la compatibilidad del lenguaje de delegados.

Compatibilidad del lenguaje para eventos

La sintaxis para definir eventos, y suscribirse o cancelar la suscripción de eventos, es una extensión de la sintaxis de los delegados.

Para definir un evento, use la palabra clave `event`:

```
public event EventHandler<FileListArgs> Progress;
```

El tipo del evento (`EventHandler<FileListArgs>` en este ejemplo) debe ser un tipo de delegado. Existen varias convenciones que debe seguir al declarar un evento. Normalmente, el tipo de delegado de eventos tiene un valor devuelto void. Las declaraciones de eventos deben ser un verbo o una frase verbal. Use un tiempo verbal pasado

cuando el evento notifique algo que ha ocurrido. Use un tiempo verbal presente (por ejemplo, `Closing`) para notificar algo que está a punto de suceder. A menudo, el uso del tiempo presente indica que su clase admite algún tipo de comportamiento de personalización. Uno de los escenarios más comunes es admitir la cancelación. Por ejemplo, un evento `Closing` puede incluir un argumento que indicará si la operación de cierre debe continuar o no. Otros escenarios pueden permitir que los autores de la llamada modifiquen el comportamiento actualizando propiedades de los argumentos de eventos. Puede generar un evento para indicar la siguiente acción propuesta que realizará un algoritmo. El controlador de eventos puede exigir una acción diferente modificando las propiedades del argumento de eventos.

Cuando quiera generar el evento, llame a los controladores de eventos mediante la sintaxis de invocación del delegado:

```
Progress?.Invoke(this, new FileListArgs(file));
```

Como se ha tratado en la sección sobre delegados, el operador ?. hace que se garantice más fácilmente que no intenta generar el evento cuando no existen suscriptores en este.

Se suscribe a un evento con el operador `+=`:

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

El método de controlador normalmente tiene el prefijo "On" seguido del nombre del evento, como se ha mostrado anteriormente.

Cancela la suscripción con el operador `-=`:

```
fileLister.Progress -= onProgress;
```

Es importante tener en cuenta que he declarado una variable local para la expresión que representa el controlador de eventos. Eso garantiza que la cancelación de la suscripción quita el controlador. Si, en su lugar, ha usado el cuerpo de la expresión lambda, está intentando quitar un controlador que nunca ha estado asociado, lo que no produce ninguna acción.

En el artículo siguiente, obtendrá más información sobre los modelos de eventos típicos y las diferentes variaciones de este ejemplo.

Siguiente

Patrón de eventos estándar de .NET

18/03/2020 • 15 minutes to read • Edit Online

Anterior

Los eventos de .NET generalmente siguen unos patrones conocidos. Estandarizar sobre estos patrones significa que los desarrolladores pueden aprovechar el conocimiento de esos patrones estándar, que se pueden aplicar a cualquier programa de evento de .NET.

Vamos a analizar los patrones estándar, para que tenga todos los conocimientos necesarios para crear orígenes de eventos estándar y suscribirse y procesar eventos estándar en el código.

Firmas de delegado de eventos

La firma estándar de un delegado de eventos de .NET es:

```
void OnEventRaised(object sender, EventArgs args);
```

El tipo de valor devuelto es void. Los eventos se basan en delegados y son delegados de multidifusión. Eso admite varios suscriptores para cualquier origen de eventos. El único valor devuelto de un método no escala a varios suscriptores de eventos. ¿Qué valor devuelto ve el origen de evento después de generar un evento? Más adelante en este artículo verá cómo crear protocolos de evento que admiten suscriptores de eventos que notifican información al origen del evento.

La lista de argumentos contiene dos argumentos: el remitente y los argumentos del evento. El tipo de tiempo de compilación de `sender` es `System.Object`, aunque probablemente conozca un tipo más derivado que siempre será correcto. Por convención, use `object`.

Típicamente, el segundo argumento era un tipo que se derivaba de `System.EventArgs`. (Verá en la siguiente sección que ya no se aplica esta convención). Si el tipo de evento no necesita ningún argumento adicional, aún tendrá que proporcionar los dos argumentos. Hay un valor especial, `EventArgs.Empty`, que debe usarse para indicar que el evento no contiene ninguna información adicional.

Vamos a crear una clase que enumera los archivos en un directorio, o cualquiera de sus subdirectorios que siguen un patrón. Este componente genera un evento para cada archivo encontrado que coincida con el modelo.

El uso de un modelo de eventos proporciona algunas ventajas de diseño. Se pueden crear varios agentes de escucha de eventos que realicen acciones diferentes cuando se encuentre un archivo buscado. La combinación de los distintos agentes de escucha puede crear algoritmos más sólidos.

Esta es la declaración del argumento de evento inicial para buscar un archivo buscado:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Aunque este tipo parece un tipo pequeño exclusivo para datos, debe seguir la convención y convertirlo en un tipo

de referencia (`class`). Esto significa que el objeto de argumento se pasará por referencia y que todos los suscriptores verán las actualizaciones de los datos. La primera versión es un objeto inmutable. Es preferible hacer que las propiedades en el tipo de argumento de evento sean inmutables. De ese modo, un suscriptor no puede cambiar los valores antes de que los vea otro suscriptor. (Hay excepciones, como verá a continuación).

Después, debemos crear la declaración de evento en la clase FileSearcher. Aprovechando el tipo `EventHandler<T>`, no es necesario crear otra definición de tipo más. Simplemente se puede usar una especialización genérica.

Vamos a rellenar la clase FileSearcher para buscar archivos que coincidan con un patrón y generar el evento correcto cuando se detecte una coincidencia.

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileFoundArgs(file));
        }
    }
}
```

Definición y generación de eventos como si fueran campos

La manera más sencilla de agregar un evento a la clase es declarar ese evento como un campo público, como en el ejemplo anterior:

```
public event EventHandler<FileFoundArgs> FileFound;
```

Parece que se está declarando un campo público, lo que podría parecer una práctica orientada a objetos incorrecta. Quiere proteger el acceso a los datos a través de propiedades o métodos. Aunque esto puede parecer una mala práctica, el código generado por el compilador crea contenedores para que solo se pueda acceder de forma segura a los objetos de evento. Las únicas operaciones disponibles en un evento con aspecto de campo son las de agregar controlador:

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

y quitar controlador:

```
fileLister.FileFound -= onFileFound;
```

Tenga en cuenta que hay una variable local para el controlador. Si usó el cuerpo de la expresión lambda, la eliminación no funcionará correctamente. Sería una instancia diferente del delegado y, en modo silencioso, no se hace nada.

El código fuera de la clase no puede generar el evento, ni puede realizar otras operaciones.

Devolución de valores desde los suscriptores de eventos

La versión simple funciona correctamente. Vamos a agregar otra característica: la cancelación.

Cuando se genera el evento encontrado, los agentes de escucha deberían ser capaces de detener el procesamiento, si este archivo es el último que se busca.

Los controladores de eventos no devuelven un valor, por lo que se necesita comunicarlo de otra forma. El patrón de eventos estándar usa el objeto EventArgs para incluir campos que los suscriptores de eventos pueden usar para comunicar la cancelación.

Existen dos patrones diferentes que podrían usarse, basándose en la semántica del contrato de cancelación. En ambos casos, se agrega un campo booleano a EventArguments para el evento del archivo encontrado.

Uno de los patrones permitiría a cualquier suscriptor cancelar la operación. Para este patrón, el nuevo campo se inicializa en `false`. Los suscriptores pueden cambiarlo a `true`. Después de que todos los suscriptores hayan visto el evento generado, el componente FileSearcher examina el valor booleano y toma medidas.

El segundo patrón solo debería cancelar la operación si todos los suscriptores quieren que se cancele. En este patrón, el nuevo campo se inicializa para indicar que se debe cancelar la operación y cualquier suscriptor puede modificarlo para indicar que la operación debe continuar. Después de que todos los suscriptores hayan visto el evento generado, el componente FileSearcher examina el valor booleano y toma medidas. Hay un paso adicional en este patrón: el componente necesita saber si los suscriptores vieron el evento. Si no hay ningún suscriptor, el campo indicaría incorrectamente una cancelación.

Vamos a implementar la primera versión de este ejemplo. Debe agregar un campo booleano denominado `CancelRequested` al tipo `FileFoundArgs`:

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set;}

    public FileFoundArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

Este nuevo campo se inicializa automáticamente en `false`, el valor predeterminado de un campo booleano, por lo que no se cancela accidentalmente. El otro cambio en el componente consiste en comprobar el indicador después de generar el evento para ver si alguno de los suscriptores solicitó una cancelación:

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

Una ventaja de este patrón es que no supone un cambio brusco. Ninguno de los suscriptores solicitó una cancelación antes y siguen sin hacerlo. No debe actualizarse el código de ningún suscriptor a menos que quieran admitir el nuevo protocolo de cancelación. Está acoplado muy holgadamente.

Vamos a actualizar el suscriptor para que solicite una cancelación una vez que encuentra el primer ejecutable:

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Agregar otra declaración de evento

Vamos a agregar una característica más y demostrar otras expresiones de lenguaje para los eventos. Vamos a agregar una sobrecarga del método `Search` que recorre todos los subdirectorios en busca de archivos.

Podría llegar a ser una operación de larga duración si el directorio tuviese muchos subdirectorios. Vamos a agregar un evento que se genera cuando comienza cada nueva búsqueda en el directorio. Esto permite a los suscriptores realizar el seguimiento y actualizar al usuario sobre el progreso. Todos los ejemplos creados hasta ahora son públicos. Convertiremos este evento en un evento interno. Eso significa que también se pueden convertir en internos los tipos que se usan para los argumentos.

Comenzará creando la nueva clase derivada de EventArgs para informar del nuevo directorio y del progreso.

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

De nuevo, puede seguir las recomendaciones para crear un tipo de referencia inmutable para los argumentos de evento.

Después, defina el evento. Esta vez, usará una sintaxis diferente. Además de usar la sintaxis de campos, puede crear explícitamente la propiedad con controladores add y remove. En este ejemplo, no necesitará código adicional en los controladores, pero aquí se muestra cómo se crean.

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { directoryChanged += value; }
    remove { directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs> directoryChanged;
```

En muchos aspectos, el código que se escribe aquí refleja el código que genera el compilador para las definiciones de evento de campo que se vieron anteriormente. El evento se crea mediante una sintaxis muy similar a la que se usó para las propiedades. Tenga en cuenta que los controladores tienen nombres diferentes: `add` y `remove`. Se llaman para suscribirse al evento o para cancelar la suscripción al evento. Tenga en cuenta que también debe declarar un campo de respaldo privado para almacenar la variable de evento. Se inicializa en null.

Después, se agregará la sobrecarga del método `Search` que recorre los subdirectorios y genera los dos eventos. La manera más fácil de hacerlo consiste en usar un argumento predeterminado para especificar que se quiere buscar en todos los directorios:

```
public void Search(string directory, string searchPattern, bool searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*", SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(directory, totalDirs, completedDirs++));
        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

En este punto, puede ejecutar la aplicación mediante la llamada a la sobrecarga para buscar en todos los subdirectorios. No hay ningún suscriptor en el nuevo evento `ChangeDirectory`, pero al usar el elemento `?.Invoke()` se garantiza que esto funciona correctamente.

Vamos a agregar un controlador para escribir una línea que muestre el progreso en la ventana de la consola.

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs} completed...");
};
```

Ha visto los patrones que se siguen en todo el ecosistema de. NET. El aprendizaje de estos patrones y convenciones le permitirá escribir elementos de C# y .NET rápidamente.

Más adelante verá algunos cambios en estos patrones en la versión más reciente de. NET.

Siguiente

Patrón de eventos actualizado de .NET Core

18/03/2020 • 7 minutes to read • Edit Online

Anterior

En el artículo anterior se describían los patrones de eventos más comunes. .NET Core tiene un patrón menos estricto. En esta versión, la definición `EventHandler<TEventArgs>` ya no tiene la restricción que obliga a que `TEventArgs` sea una clase derivada de `System.EventArgs`.

Esto aumenta la flexibilidad y es compatible con versiones anteriores. Comencemos con la flexibilidad. La clase System.EventArgs introduce un método, `MemberwiseClone()`, que crea una copia superficial del objeto. Dicho método debe usar la reflexión para implementar su función en cualquier clase derivada de `EventArgs`. Esta funcionalidad es más fácil de crear en una clase derivada concreta. Esto significa que derivar de System.EventArgs es una restricción que limita los diseños, pero no proporciona ninguna ventaja adicional. De hecho, puede cambiar las definiciones de `FileFoundArgs` y `SearchDirectoryArgs` para que no deriven de `EventArgs`. El programa funcionará exactamente igual.

También puede cambiar `SearchDirectoryArgs` a un struct si realiza un cambio más:

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

El cambio adicional consiste en llamar al constructor sin parámetros antes de entrar en el constructor que inicializa todos los campos. Sin esta adición, las reglas de C# informarán de que se está teniendo acceso a las propiedades antes de que se hayan asignado.

No debe cambiar `FileFoundArgs` de una clase (tipo de referencia) a un struct (tipo de valor). Esto se debe a que el protocolo para controlar la cancelación requiere que los argumentos de evento se pasen por referencia. Si realizase el mismo cambio, la clase de búsqueda de archivos no podría observar nunca los cambios realizados por ninguno de los suscriptores de eventos. Se usaría una nueva copia de la estructura para cada suscriptor, y dicha copia sería diferente de la que ve el objeto de búsqueda de archivos.

Ahora veamos cómo este cambio puede ser compatible con versiones anteriores. La eliminación de la restricción no afecta al código existente. Los tipos de argumento de evento existentes siguen derivando de `System.EventArgs`. La compatibilidad con versiones anteriores es uno de los motivos principales por los que siguen derivando de `System.EventArgs`. Los suscriptores de eventos existentes serán suscriptores a un evento que haya seguido el patrón clásico.

Según una lógica similar, cualquier tipo de argumento de evento creado ahora no tendría ningún suscriptor en el código base existente. Los nuevos tipos de evento que no deriven de `System.EventArgs` no interrumpirán ese código base.

Eventos con suscriptores Async

Le queda un último patrón que aprender: Cómo escribir correctamente suscriptores de eventos que llaman a código asincrónico. Este reto se describe en el artículo sobre async y await. Los métodos asincrónicos pueden tener un tipo de valor devuelto void, pero esto no es recomendable. Cuando el código de suscriptor de eventos llama a un método asincrónico, no le queda otra opción que crear un método `async void`, ya que lo requiere la firma del controlador de eventos.

Debe conciliar estas instrucciones contradictorias. De alguna manera, debe crear un método `async void` seguro. A continuación se muestran los aspectos básicos del patrón que debe implementar:

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

En primer lugar, observe que el controlador está marcado como un controlador asincrónico. Dado que se va a asignar a un tipo de delegado de controlador de eventos, tendrá un tipo de valor devuelto void. Esto significa que debe seguir el patrón que se muestra en el controlador y no debe permitir que se produzca ninguna excepción fuera del contexto del controlador asincrónico. Como no devuelve una tarea, no hay ninguna tarea que pueda notificar el error entrando en el estado de error. Dado que el método es asincrónico, no puede producir la excepción. (El método de llamada ha continuado con la ejecución porque es `async`). El comportamiento real en tiempo de ejecución se definirá de forma diferente para diferentes entornos. Se puede terminar el subproceso o el proceso que posee el subproceso, o dejar el proceso en un estado indeterminado. Todas estas salidas potenciales son altamente no deseables.

Por eso debe encapsular la instrucción await para la tarea asincrónica en su propio bloque try. Si esto genera una tarea con error, puede registrar el error. Si se produce un error del que no se puede recuperar la aplicación, puede salir del programa de forma rápida y correctamente.

Estas son las principales actualizaciones del patrón de eventos de .NET. Verá numerosos ejemplos de las versiones anteriores de las bibliotecas con las que trabaje. Aun así, también debe entender los patrones más recientes.

El siguiente artículo de esta serie le ayudará a distinguir entre el uso de `delegates` y `events` en los diseños. Dado que se trata de conceptos similares, el artículo le ayudará a tomar la mejor decisión para sus programas.

Siguiente

Distinción de delegados y eventos

16/04/2020 • 7 minutes to read • Edit Online

Anterior

Los desarrolladores que son nuevos en la plataforma de NET Core a menudo tienen problemas para decidir entre un diseño basado en `delegates` y uno basado en `events`. La elección de delegados o eventos suele ser difícil, ya que las dos características de lenguaje son similares. Los eventos incluso se crean con compatibilidad de lenguaje para los delegados.

Ambos ofrecen un escenario de enlace en tiempo de ejecución: permiten escenarios donde un componente se comunica mediante una llamada a un método que solo se conoce en tiempo de ejecución. Ambos admiten métodos de suscriptor único y múltiple. Puede que se haga referencia a estos términos como compatibilidad con multidifusión o de conversión única. Ambos admiten una sintaxis similar para agregar y quitar controladores. Por último, para generar un evento y llamar a un delegado se usa exactamente la misma sintaxis de llamada de método. Incluso los dos admiten la misma sintaxis del método `Invoke()` para su uso con el operador `?.`.

Con todas estas similitudes, es fácil tener problemas para determinar cuándo usar cada uno.

La escucha de eventos es opcional

La consideración más importante para determinar qué característica de lenguaje usar es si debe haber o no un suscriptor adjunto. Si el código debe llamar al código proporcionado por el suscriptor, debe usar un diseño basado en delegados. Si el código puede completar todo su trabajo sin llamar a ningún suscriptor, debe usar un diseño basado en eventos.

Tenga en cuenta los ejemplos que se crean en esta sección. Al código que ha creado con `List.Sort()` se le debe proporcionar una función de comparador para ordenar los elementos de manera adecuada. Las consultas LINQ deben proporcionarse con delegados para determinar qué elementos se van a devolver. Ambos han usado un diseño creado con delegados.

Considere el evento `Progress`. Notifica el progreso de una tarea. La tarea continúa haya o no agentes de escucha. `FileSearcher` es otro ejemplo. Todavía buscaría y encontraría todos los archivos que se han solicitado, incluso sin ningún suscriptor de eventos adjunto. Los controles de UX todavía funcionan correctamente, incluso cuando no hay ningún suscriptor escuchando los eventos. Ambos usan diseños basados en eventos.

Los valores devueltos necesitan delegados

Otra consideración es el prototipo del método que quiere para el método delegado. Como ha visto, todos los delegados que se han usado para los eventos tienen un tipo de valor devuelto void. También ha visto que hay elementos para crear controladores de eventos que pasan información de nuevo a los orígenes de eventos mediante la modificación de las propiedades del objeto de argumento del evento. Aunque estos elementos funcionan, no son tan naturales como la devolución de un valor de un método.

Observe que estas dos heurísticas suelen estar presentes: si el método delegado devuelve un valor, lo más probable es que afecte de algún modo al algoritmo.

Los eventos tienen invocación privada

Las clases distintas de la clase en la que se encuentra un evento solo pueden agregar y quitar clientes de escucha de eventos; solo la clase que contiene el evento puede invocarlo. Los eventos suelen ser miembros de clase

públicos. En cambio, a menudo los delegados se pasan como parámetros y se almacenan como miembros de clase privada si no están almacenados.

Los agentes de escucha de eventos a menudo tienen una vigencia mayor

El hecho de que esos clientes de escucha de eventos tengan una duración más larga es una justificación ligeramente más débil. En cambio, puede encontrar que los diseños basados en eventos son más naturales cuando el origen de eventos generará eventos durante un período de tiempo largo. Puede ver ejemplos de diseño basado en eventos para los controles de la experiencia del usuario en muchos sistemas. Cuando se suscriba a un evento, el origen de eventos puede generar eventos durante la vigencia del programa. (Puede anular la suscripción de los eventos cuando ya no los necesite).

Compare eso con muchos diseños basados en delegados, donde un delegado se usa como un argumento para un método, y el delegado no se usa después de que se devuelva ese método.

Evaluar cuidadosamente

Las consideraciones anteriores no son reglas rápidas ni estrictas. En su lugar, representan instrucciones que pueden ayudarle a decidir qué opción es mejor para su uso particular. Como son similares, incluso puede crear un prototipo de los dos y considerar con cuál sería más natural trabajar. Ambos controlan escenarios de enlace en tiempo de ejecución correctamente. Use el que comunique mejor su diseño.

Language-Integrated Query (LINQ)

18/03/2020 • 6 minutes to read • Edit Online

Language-Integrated Query (LINQ) es el nombre de un conjunto de tecnologías basadas en la integración de capacidades de consulta directamente en el lenguaje C#. Tradicionalmente, las consultas con datos se expresaban como cadenas simples sin comprobación de tipos en tiempo de compilación ni compatibilidad con IntelliSense. Además, tiene que aprender un lenguaje de consultas diferente para cada tipo de origen de datos: bases de datos SQL, documentos XML y varios servicios web, entre otros. Con LINQ, una consulta es una construcción de lenguaje de primera clase, como clases, métodos y eventos.

Para un desarrollador que escribe consultas, la parte más visible de "lenguaje integrado" de LINQ es la expresión de consulta. Las expresiones de consulta se escriben con una *sintaxis de consulta* declarativa. Con la sintaxis de consulta, puede realizar operaciones de filtrado, ordenación y agrupamiento en orígenes de datos con el mínimo código. Utilice los mismos patrones de expresión de consulta básica para consultar y transformar datos de bases de datos SQL, conjuntos de datos de ADO .NET, secuencias y documentos XML y colecciones. NET.

En el ejemplo siguiente se muestra la operación de consulta completa. La operación completa incluye crear un origen de datos, definir la expresión de consulta y ejecutar la consulta en una instrucción `foreach`.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

Información general sobre la expresión de consulta

- Las expresiones de consulta se pueden utilizar para consultar y transformar los datos de cualquier origen de datos habilitado para LINQ. Por ejemplo, una sola consulta puede recuperar datos de una base de datos SQL y generar una secuencia XML como salida.
- Las expresiones de consulta son fáciles de controlar porque utilizan muchas construcciones de lenguaje C# familiares.
- Todas las variables de una expresión de consulta están fuertemente tipadas, aunque en muchos

casos no es necesario proporcionar el tipo explícitamente porque el compilador puede deducirlo. Para más información, vea Relaciones entre tipos en operaciones de consulta LINQ.

- Una consulta no se ejecuta hasta que no se realiza la iteración a través de la variable de consulta, por ejemplo, en una instrucción `foreach`. Para obtener más información, vea Introduction to LINQ queries (Introducción a las consultas LINQ).

- En tiempo de compilación, las expresiones de consulta se convierten en llamadas al método de operador de consulta estándar según las reglas establecidas en la especificación de C#. Cualquier consulta que se puede expresar con sintaxis de consulta también se puede expresar mediante sintaxis de método. Sin embargo, en la mayoría de los casos, la sintaxis de consulta es más legible y concisa. Para más información, vea Especificación del lenguaje C# e Información general sobre operadores de consulta estándar.

- Como regla al escribir consultas LINQ, se recomienda utilizar la sintaxis de consulta siempre que sea posible y la sintaxis de método cuando sea necesario. No hay diferencias semánticas ni de rendimiento entre estas dos formas diversas. Las expresiones de consulta suelen ser más legibles que las expresiones equivalentes escritas con la sintaxis de método.

- Algunas operaciones de consulta, como Count o Max, no tienen ninguna cláusula de expresión de consulta equivalente, de modo que deben expresarse como una llamada de método. La sintaxis de método se puede combinar con la sintaxis de consulta de varias maneras. Para más información, vea Sintaxis de consultas y sintaxis de métodos en LINQ.

- Las expresiones de consulta pueden compilarse en árboles de expresión o en delegados, en función del tipo al que se aplica la consulta. Las consultas IEnumerable<T> se compilan en delegados. Las consultas IQueryable y IQueryable<T> se compilan en árboles de expresión. Para más información, vea Expression trees (Árboles de expresión).

Pasos siguientes

Para obtener más información sobre LINQ, empiece a familiarizarse con algunos conceptos básicos en Conceptos básicos de las expresiones de consultas y, después, lea la documentación de la tecnología de LINQ en la que esté interesado:

- Documentos XML: LINQ to XML (LINQ para XML)

- ADO.NET Entity Framework: LINQ to entities (LINQ para entidades)

- Colecciones .NET, archivos y cadenas, entre otros: LINQ to objects (LINQ para objetos)

Para comprender mejor los aspectos generales de LINQ, vea LINQ in C# (LINQ en C#).

Para empezar a trabajar con LINQ en C#, vea el tutorial Trabajar con LINQ.

Conceptos básicos de las expresiones de consultas

18/03/2020 • 23 minutes to read • Edit Online

En este artículo se presentan los conceptos básicos relacionados con las expresiones de consulta en C#.

¿Qué es una consulta y qué hace?

Una *consulta* es un conjunto de instrucciones que describen qué datos se recuperan de uno o varios orígenes de datos determinados y qué forma y qué organización deben tener los datos devueltos. Una consulta es distinta de los resultados que genera.

Por lo general, los datos de origen se organizan lógicamente como una secuencia de elementos del mismo tipo. Por ejemplo, una tabla de base de datos SQL contiene una secuencia de filas. En un archivo XML, hay una "secuencia" de elementos XML (aunque estos se organizan jerárquicamente en una estructura de árbol). Una colección en memoria contiene una secuencia de objetos.

Desde el punto de vista de la aplicación, el tipo y la estructura específicos de los datos de origen originales no es importante. La aplicación siempre ve los datos de origen como una colección IEnumerable<T> o IQueryable<T>. Por ejemplo, en LINQ to XML, los datos de origen se hacen visibles como `IEnumerable`<XElement>.

Dada esta secuencia de origen, una consulta puede hacer una de estas tres cosas:

- Recuperar un subconjunto de los elementos para generar una nueva secuencia sin modificar los elementos individuales. Después, la consulta puede ordenar o agrupar la secuencia devuelta de varias maneras, como se muestra en el ejemplo siguiente (supongamos que `scores` es `int[]`):

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Recuperar una secuencia de elementos como en el ejemplo anterior, pero transformándolos en un nuevo tipo de objeto. Por ejemplo, una consulta puede recuperar solo los apellidos de ciertos registros de clientes de un origen de datos. También puede recuperar el registro completo y, luego, usarlo para construir otro tipo de objeto en memoria, o incluso datos XML, antes de generar la secuencia de resultado final. En el ejemplo siguiente muestra una proyección de `int` a `string`. Observe el nuevo tipo de `highScoresQuery`.

  ```
  IEnumerable<string> highScoresQuery2 =
      from score in scores
      where score > 80
      orderby score descending
      select $"The score is {score}";
  ```

- Recuperar un valor singleton sobre los datos de origen, por ejemplo:
  - El número de elementos que coinciden con una condición determinada.
  - El elemento que tiene el mayor o el menor valor.
  - El primer elemento que coincide con una condición, o bien la suma de determinados valores de un conjunto de elementos especificado. Por ejemplo, la consulta siguiente devuelve el número de resultados mayor que 80 de la matriz de enteros `scores`:

```
int highScoreCount =
    (from score in scores
     where score > 80
     select score)
     .Count();
```

En el ejemplo anterior, observe el uso de los paréntesis alrededor de la expresión de consulta antes de llamar al método `Count`. Esto también se puede expresar mediante una nueva variable para almacenar el resultado concreto. Esta técnica es más legible porque hace que la variable que almacena la consulta se mantenga separada de la consulta que almacena un resultado.

```
IEnumerable<int> highScoresQuery3 =
    from score in scores
    where score > 80
    select score;

int scoreCount = highScoresQuery3.Count();
```

En el ejemplo anterior, la consulta se ejecuta en la llamada a `Count`, ya que `Count` debe iterar los resultados para determinar el número de elementos devueltos por `highScoresQuery`.

¿Qué es una expresión de consulta?

Una *expresión de consulta* es una consulta que se expresa en sintaxis de consulta. Una expresión de consulta es una construcción de lenguaje de primera clase. Es igual que cualquier otra expresión y puede usarse en cualquier contexto en el que una expresión de C# sea válida. Una expresión de consulta consta de un conjunto de cláusulas escritas en una sintaxis declarativa similar a SQL o XQuery. Cada cláusula contiene una o más expresiones de C#, y estas expresiones pueden ser una expresión de consulta en sí mismas o bien contener una expresión de consulta.

Una expresión de consulta debe comenzar con una cláusula from y debe terminar con una cláusula select o group. Entre la primera cláusula `from` y la última cláusula `select` o `group`, puede contener una o varias de estas cláusulas opcionales: where, orderby, join, let e incluso cláusulas from adicionales. También puede usar la palabra clave into para que el resultado de una cláusula `join` o `group` actúe como el origen de las cláusulas de consulta adicionales en la misma expresión de consulta.

Variable de consulta

En LINQ, una variable de consulta es cualquier variable que almacene una *consulta* en lugar de los *resultados* de una consulta. Más concretamente, una variable de consulta es siempre un tipo enumerable que generará una secuencia de elementos cuando se itere en una instrucción `foreach` o en una llamada directa a su método `IEnumerator.MoveNext`.

En el ejemplo de código siguiente se muestra una expresión de consulta simple con un origen de datos, una cláusula de filtrado, una cláusula de clasificación y ninguna transformación en los elementos de origen. La cláusula `select` finaliza la consulta.

```
static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<int> scoreQuery = //query variable
        from score in scores //required
        where score > 80 // optional
        orderby score descending // optional
        select score; //must end with select or group

    // Execute the query to produce the results
    foreach (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 93 90 82 82
```

En el ejemplo anterior, `scoreQuery` es una *variable de consulta*, que a veces se conoce simplemente como una *consulta*. La variable de consulta no almacena datos de resultado reales, que se producen en el bucle `foreach`. Cuando se ejecuta la instrucción `foreach`, los resultados de la consulta no se devuelven a través de la variable de consulta `scoreQuery`, sino a través de la variable de iteración `testScore`. La variable `scoreQuery` se puede iterar en un segundo bucle `foreach`. Siempre y cuando ni esta ni el origen de datos se hayan modificado, producirá los mismos resultados.

Una variable de consulta puede almacenar una consulta expresada en sintaxis de consulta, en sintaxis de método o en una combinación de ambas. En los ejemplos siguientes, `queryMajorCities` y `queryMajorCities2` son variables de consulta:

```
//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population > 100000);
```

Por otro lado, en los dos ejemplos siguientes se muestran variables que no son de consulta, a pesar de que se inicialicen con una consulta. No son variables de consulta porque almacenan resultados:

```
int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> largeCitiesList =
    (from country in countries
     from city in country.Cities
     where city.Population > 10000
     select city)
       .ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();
```

Para obtener más información sobre las distintas formas de expresar consultas, vea Query syntax and method syntax in LINQ (Sintaxis de consulta y sintaxis de método en LINQ).

Asignación implícita y explícita de tipos de variables de consulta

En esta documentación se suele proporcionar el tipo explícito de la variable de consulta para mostrar las relaciones de tipo entre la variable de consulta y la cláusula select. Pero también se puede usar la palabra clave var para indicarle al compilador que infiera el tipo de una variable de consulta (u otra variable local) en tiempo de compilación. Por ejemplo, la consulta de ejemplo que se mostró anteriormente en este tema también se puede expresar mediante la asignación implícita de tipos:

```
// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

Para obtener más información, vea Implicitly typed local variables (Variables locales con asignación implícita de tipos) y Type relationships in LINQ query operations (Relaciones entre tipos en las operaciones de consulta de LINQ).

Iniciar una expresión de consulta

Una expresión de consulta debe comenzar con una cláusula `from`, que especifica un origen de datos junto con una variable de rango. La variable de rango representa cada elemento sucesivo de la secuencia de origen a medida que esta se recorre. La variable de rango está fuertemente tipada en función del tipo de elementos del origen de datos. En el ejemplo siguiente, como `countries` es una matriz de objetos `Country`, la variable de rango también está tipada como `Country`. Dado que la variable de rango está fuertemente tipada, se puede usar el operador punto para tener acceso a cualquier miembro disponible del tipo.

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 500000 //sq km
    select country;
```

La variable de rango está en el ámbito hasta que se cierra la consulta con un punto y coma o con una cláusula de *continuación*.

Una expresión de consulta puede contener varias cláusulas `from`. Use más cláusulas `from` cuando cada elemento de la secuencia de origen sea una colección en sí mismo o contenga una colección. Por ejemplo, supongamos que tiene una colección de objetos `Country`, cada uno de los cuales contiene una colección de objetos `City` denominados `Cities`. Para consultar los objetos `City` de cada `Country`, use dos cláusulas `from`, como se muestra aquí:

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

Para obtener más información, vea from clause (Cláusula from).

Finalizar una expresión de consulta

Una expresión de consulta debe finalizar con una cláusula `group` o `select`.

group (cláusula)

Use la cláusula `group` para generar una secuencia de grupos organizados por la clave que especifique. La clave puede ser cualquier tipo de datos. Por ejemplo, la siguiente consulta crea una secuencia de grupos que contienen uno o más objetos `Country` y cuya clave es un valor `char`.

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

Para obtener más información sobre la agrupación, vea group clause (Cláusula group).

select (cláusula)

Use la cláusula `select` para generar todos los demás tipos de secuencias. Una cláusula `select` simple solo genera una secuencia del mismo tipo de objetos que los objetos contenidos en el origen de datos. En este ejemplo, el origen de datos contiene objetos `Country`. La cláusula `orderby` simplemente ordena los elementos con un orden nuevo y la cláusula `select` genera una secuencia con los objetos `Country` reordenados.

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

La cláusula `select` puede usarse para transformar los datos de origen en secuencias de nuevos tipos. Esta transformación también se denomina *proyección*. En el ejemplo siguiente, la cláusula `select` *proyecta* una secuencia de tipos anónimos que solo contiene un subconjunto de los campos del elemento original. Tenga en cuenta que los nuevos objetos se inicializan mediante un inicializador de objeto.

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

Para obtener más información sobre todas las formas en que se puede usar una cláusula `select` para transformar datos de origen, vea select clause (Cláusula select).

Continuaciones con "into"

Puede usar la palabra clave `into` en una cláusula `select` o `group` para crear un identificador temporal que almacene una consulta. Hágalo cuando deba realizar operaciones de consulta adicionales en una consulta después de una operación de agrupación o selección. En el siguiente ejemplo se agrupan los objetos `countries` según su población en intervalos de 10 millones. Una vez que se han creado estos grupos, las cláusulas adicionales filtran algunos grupos y, después, ordenan los grupos en orden ascendente. Para realizar esas operaciones adicionales, es necesaria la continuación representada por `countryGroup`.

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int) country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
        Console.WriteLine(country.Name + ":" + country.Population);
}
```

Para obtener más información, vea into.

Filtrar, ordenar y combinar

Entre la cláusula de inicio `from` y la cláusula de finalización `select` o `group`, todas las demás cláusulas (`where`, `join`, `orderby`, `from`, `let`) son opcionales. Cualquiera de las cláusulas opcionales puede usarse cero o varias veces en el cuerpo de una consulta.

where (cláusula)

Use la cláusula `where` para filtrar los elementos de los datos de origen en función de una o varias expresiones de predicado. La cláusula `where` del ejemplo siguiente tiene un predicado con dos condiciones.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 100000
    select city;
```

Para obtener más información, vea where (Cláusula).

orderby (cláusula)

Use la cláusula `orderby` para ordenar los resultados en orden ascendente o descendente. También puede especificar criterios de ordenación secundaria. En el ejemplo siguiente se realiza una ordenación primaria de los objetos `country` mediante la propiedad `Area`. Después, se realiza una ordenación secundaria mediante la propiedad `Population`.

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

La palabra clave `ascending` es opcional; es el criterio de ordenación predeterminado si no se especifica ningún orden. Para obtener más información, vea orderby (Cláusula).

join (cláusula)

Use la cláusula `join` para asociar o combinar elementos de un origen de datos con elementos de otro origen de datos en función de una comparación de igualdad entre las claves especificadas en cada elemento. En LINQ, las operaciones de combinación se realizan en secuencias de objetos cuyos elementos son de tipos diferentes. Después de combinar dos secuencias, debe usar una instrucción `select` o `group` para especificar qué elemento se va a almacenar en la secuencia de salida. También puede usar un tipo anónimo para combinar propiedades de cada conjunto de elementos asociados en un nuevo tipo para la secuencia de salida. En el ejemplo siguiente se asocian objetos `prod` cuya propiedad `Category` coincide con una de las categorías de la matriz de cadenas `categories`. Los productos cuya propiedad `Category` no coincide con ninguna cadena de `categories` se filtran. La instrucción `select` proyecta un nuevo tipo cuyas propiedades se toman de `cat` y `prod`.

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new { Category = cat, Name = prod.Name };
```

También puede realizar una combinación agrupada. Para ello, almacene los resultados de la operación `join` en una variable temporal mediante el uso de la palabra clave into. Para obtener más información, vea join (Cláusula, Referencia de C#).

let (cláusula)

Use la cláusula `let` para almacenar el resultado de una expresión, como una llamada de método, en una nueva variable de rango. En el ejemplo siguiente, la variable de rango `firstName` almacena el primer elemento de la matriz de cadenas devuelta por `Split`.

```
string[] names = { "Svetlana Omelchenko", "Claire O'Donnell", "Sven Mortensen", "Cesar Garcia" };
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (string s in queryFirstNames)
    Console.Write(s + " ");
//Output: Svetlana Claire Sven Cesar
```

Para obtener más información, vea let (Cláusula).

Subconsultas en una expresión de consulta

Una cláusula de consulta puede contener una expresión de consulta, en ocasiones denominada *subconsulta*. Cada subconsulta comienza con su propia cláusula `from` que no necesariamente hace referencia al mismo origen de datos de la primera cláusula `from`. Por ejemplo, la consulta siguiente muestra una expresión de consulta que se usa en la instrucción select para recuperar los resultados de una operación de agrupación.

```
var queryGroupMax =
    from student in students
    group student by student.GradeLevel into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore =
            (from student2 in studentGroup
             select student2.Scores.Average())
             .Max()
    };
```

Para más información, consulte Realizar una subconsulta en una operación de agrupación.

Vea también

- Guía de programación de C#
- Language-Integrated Query (LINQ)
- Palabras clave de consultas (LINQ)
- Información general sobre operadores de consulta estándar

LINQ en C#

18/03/2020 • 2 minutes to read • Edit Online

Esta sección contiene vínculos a temas que ofrecen información más detallada sobre LINQ.

En esta sección

Introducción a las consultas LINQ
Describe las tres partes de la operación de consulta LINQ básica comunes a todos los lenguajes y orígenes de datos.

LINQ y tipos genéricos
Ofrece una breve introducción a los tipos genéricos, tal como se usan en LINQ.

Transformaciones de datos con LINQ
Describe las diversas maneras de transformar datos recuperados en las consultas.

Relaciones entre tipos en las operaciones de consulta LINQ
Describe cómo se mantienen o transforman los tipos en las tres partes de una operación de consulta LINQ.

Sintaxis de consulta y sintaxis de método en LINQ
Compara la sintaxis de método y la sintaxis de consulta como dos maneras de expresar una consulta LINQ.

Características de C# compatibles con LINQ
Describe las construcciones de lenguaje de C# compatibles con LINQ.

Secciones relacionadas

Expresiones de consulta LINQ
Incluye información general sobre las consultas en LINQ y proporciona vínculos a recursos adicionales.

Información general sobre operadores de consulta estándar
Presenta los métodos estándar usados en LINQ.

Escribir consultas LINQ en C#

18/03/2020 • 7 minutes to read • Edit Online

En este artículo se muestran las tres formas de escribir una consulta LINQ en C#:

1. Usar sintaxis de consulta.
2. Usar sintaxis de método.
3. Usar una combinación de sintaxis de consulta y sintaxis de método.

Los ejemplos siguientes muestran algunas consultas LINQ sencillas mediante cada enfoque enumerado anteriormente. En general, la regla es usar (1) siempre que sea posible y usar (2) y (3) cuando sea necesario.

> **NOTE**
> Estas consultas funcionan en colecciones en memoria simples, pero la sintaxis básica es idéntica a la empleada en LINQ to Entities y LINQ to XML.

Ejemplo: Sintaxis de consulta

La manera recomendada de escribir la mayoría de las consultas es usar *sintaxis de consulta* para crear *expresiones de consulta*. En el siguiente ejemplo se muestran tres expresiones de consulta. La primera expresión de consulta muestra cómo filtrar o restringir los resultados mediante la aplicación de condiciones con una cláusula `where`. Devuelve todos los elementos de la secuencia de origen cuyos valores sean mayores que 7 o menores que 3. La segunda expresión muestra cómo ordenar los resultados devueltos. La tercera expresión muestra cómo agrupar los resultados según una clave. Esta consulta devuelve dos grupos en función de la primera letra de la palabra.

```
// Query #1.
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

// The query variable can also be implicitly typed by using var
IEnumerable<int> filteringQuery =
    from num in numbers
    where num < 3 || num > 7
    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num < 3 || num > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = { "carrots", "cabbage", "broccoli", "beans", "barley" };
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

Tenga en cuenta que el tipo de las consultas es IEnumerable<T>. Todas estas consultas podrían escribirse mediante `var` como se muestra en el ejemplo siguiente:

```
var query = from num in numbers...
```

En cada ejemplo anterior, las consultas no se ejecutan realmente hasta que se recorre en iteración la variable de consulta en una instrucción `foreach` o cualquier otra instrucción. Para más información, vea Introduction to LINQ Queries (Introducción a las consultas LINQ).

Ejemplo: Sintaxis de método

Algunas operaciones de consulta deben expresarse como una llamada a método. Los más comunes de dichos métodos son los que devuelven valores numéricos de singleton, como Sum, Max, Min, Average y así sucesivamente. A estos métodos siempre se los debe llamar en último lugar en cualquier consulta porque representan un solo valor y no pueden servir como origen para una operación de consulta adicional. En el ejemplo siguiente se muestra una llamada a método en una expresión de consulta:

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };
// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

Si el método tiene parámetros Action o Func, se proporcionan en forma de expresión lambda, como se muestra en el ejemplo siguiente:

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

En las consultas anteriores, solo la número 4 se ejecuta inmediatamente. Esto se debe a que devuelve un valor único, y no una colección IEnumerable<T> genérica. El propio método tiene que usar `foreach` para calcular su valor.

Cada una de las consultas anteriores puede escribirse mediante tipos implícitos con var, como se muestra en el ejemplo siguiente:

```
// var is used for convenience in these queries
var average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Ejemplo: Sintaxis de consulta y método combinada

En este ejemplo se muestra cómo usar la sintaxis de método en los resultados de una cláusula de consulta. Simplemente escriba entre paréntesis la expresión de consulta y luego aplique el operador punto y llame al método. En el ejemplo siguiente la consulta número 7 devuelve un recuento de los números cuyo valor está comprendido entre 3 y 7. Pero en general es mejor usar una segunda variable para almacenar el resultado de la llamada a método. De esta manera es menos probable que la consulta se confunda con los resultados de la consulta.

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int numCount2 = numbersQuery.Count();
```

Dado que la consulta número 7 devuelve un solo valor y no una colección, se ejecuta inmediatamente.

La consulta anterior puede escribirse mediante tipos implícitos con `var` como sigue:

```
var numCount = (from num in numbers...
```

Puede escribirse en sintaxis de método como sigue:

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Puede escribirse mediante tipos explícitos como sigue:

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

Vea también

- Walkthrough: Writing Queries in C# (Tutorial: Escribir consultas en C#)
- Language-Integrated Query (LINQ)
- where (cláusula)

Consultar una colección de objetos

18/03/2020 • 3 minutes to read • Edit Online

En este ejemplo se muestra cómo realizar una consulta simple en una lista de objetos `Student`. Cada objeto `Student` contiene información básica sobre el alumno y una lista que representa las puntuaciones del alumno en cuatro exámenes.

Esta aplicación sirve de marco en muchos otros ejemplos de esta sección que usan el mismo origen de datos `students`.

Ejemplo

La consulta siguiente devuelve los alumnos que reciben una puntuación de 90 o más en su primer examen.

```
public class Student
{
    #region data
    public enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public GradeLevel Year;
    public List<int> ExamScores;

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", Id = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", Id = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", Id = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", Id = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", Id = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", Id = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", Id = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", Id = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", Id = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", Id = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", Id = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 94, 92, 91, 91}},
```

```
        new Student {FirstName = "Eugene", LastName = "Zabokritski", Id = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 96, 85, 91, 60}}
    };
    #endregion

    // Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }

    public static void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        Student.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Esta consulta es deliberadamente simple para permitirle experimentar. Por ejemplo, puede probar otras condiciones en la cláusula `where` o usar una cláusula `orderby` para ordenar los resultados.

Vea también

- Language-Integrated Query (LINQ)
- Interpolación de cadenas

Cómo devolver una consulta desde un método (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

En este ejemplo se muestra cómo devolver una consulta desde un método como un valor devuelto y como un parámetro `out`.

Los objetos de consulta admiten composición, lo que significa que puede devolver una consulta desde un método. Los objetos que representan consultas no almacenan la colección resultante, sino los pasos para generar los resultados cuando sea necesario. La ventaja de devolver objetos de consulta desde métodos es que se pueden componer o modificar todavía más. Por lo tanto, cualquier valor devuelto o parámetro `out` de un método que devuelve una consulta también debe tener ese tipo. Si un método materializa una consulta en un tipo concreto List<T> o Array, se considera que está devolviendo los resultados de la consulta en lugar de la propia consulta. Una variable de consulta que se devuelve desde un método sigue pudiendo componerse o modificarse.

Ejemplo

En el ejemplo siguiente, el primer método devuelve una consulta como un valor devuelto y el segundo método devuelve una consulta como un parámetro `out`. Tenga en cuenta que, en ambos casos, se trata de una consulta que se devuelve, no de los resultados de la consulta.

```
class MQ
{
    // QueryMethhod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }
```

```
        }

        // You also can execute the query returned from QueryMethod1
        // directly, without using myQuery1.
        Console.WriteLine("\nResults of executing myQuery1 directly:");
        // Rest the mouse pointer over the call to QueryMethod1 to see its
        // return type.
        foreach (string s in app.QueryMethod1(ref nums))
        {
            Console.WriteLine(s);
        }

        IEnumerable<string> myQuery2;
        // QueryMethod2 returns a query as the value of its out parameter.
        app.QueryMethod2(ref nums, out myQuery2);

        // Execute the returned query.
        Console.WriteLine("\nResults of executing myQuery2:");
        foreach (string s in myQuery2)
        {
            Console.WriteLine(s);
        }

        // You can modify a query by using query composition. A saved query
        // is nested inside a new query definition that revises the results
        // of the first query.
        myQuery1 = from item in myQuery1
                   orderby item descending
                   select item;

        // Execute the modified query.
        Console.WriteLine("\nResults of executing modified myQuery1:");
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Vea también

- Language-Integrated Query (LINQ)

Almacenar los resultados de una consulta en memoria

18/03/2020 • 2 minutes to read • Edit Online

Una consulta es básicamente un conjunto de instrucciones sobre cómo recuperar y organizar los datos. Las consultas se ejecutan de forma diferida, ya que se solicita cada elemento subsiguiente del resultado. Cuando se usa `foreach` para iterar los resultados, los elementos se devuelven a medida que se tiene acceso a ellos. Para evaluar una consulta y almacenar los resultados sin ejecutar un bucle `foreach`, simplemente llame a uno de los métodos siguientes en la variable de consulta:

- ToList
- ToArray
- ToDictionary
- ToLookup

Recomendamos que, al almacenar los resultados de consulta, asigne el objeto de colección devuelto a una nueva variable tal como se muestra en el ejemplo siguiente:

Ejemplo

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    static void Main()
    {

        IEnumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

Vea también

- Language-Integrated Query (LINQ)

Agrupar los resultados de consultas

18/03/2020 • 10 minutes to read • Edit Online

La agrupación es una de las capacidades más eficaces de LINQ. Los ejemplos siguientes muestran cómo agrupar datos de varias maneras:

- Por una sola propiedad.
- Por la primera letra de una propiedad de cadena.
- Por un intervalo numérico calculado.
- Por un predicado booleano u otra expresión.
- Por una clave compuesta.

Además, las dos últimas consultas proyectan sus resultados en un nuevo tipo anónimo que solo contiene el nombre y los apellidos del estudiante. Para obtener más información, vea la cláusula group.

Ejemplo

Todos los ejemplos de este tema usan los siguientes orígenes de datos y clases del asistente.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
```

```
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 96, 85, 91, 60}}
    };
    #endregion

    //Helper method, used in GroupByRange.
    protected static int GetPercentile(Student s)
    {
        double avg = s.ExamScores.Average();
        return avg > 0 ? (int)avg / 10 : 0;
    }

    public void QueryHighScores(int exam, int score)
    {
        var highScores = from student in students
                         where student.ExamScores[exam] > score
                         select new {Name = student.FirstName, Score = student.ExamScores[exam]};

        foreach (var item in highScores)
        {
            Console.WriteLine($"{item.Name,-15}{item.Score}");
        }
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Ejemplo

En el ejemplo siguiente se muestra cómo agrupar elementos de origen mediante una propiedad única del elemento como la clave de grupo. En este caso, la clave es una `string`, el apellido del alumno. También es posible usar una subcadena para la clave. La operación de agrupación usa al comparador de igualdad predeterminado para el tipo.

Pegue el método siguiente en la clase `StudentClass`. Cambie la instrucción de llamada en el método `Main` por `sc.GroupBySingleProperty()`.

```
public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by a single property in an object:
    Key: Adams
            Adams, Terry
    Key: Fakhouri
            Fakhouri, Fadi
    Key: Feng
            Feng, Hanying
    Key: Garcia
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: Mortensen
            Mortensen, Sven
    Key: O'Donnell
            O'Donnell, Claire
    Key: Omelchenko
            Omelchenko, Svetlana
    Key: Tucker
            Tucker, Lance
            Tucker, Michael
    Key: Zabokritski
            Zabokritski, Eugene
*/
```

Ejemplo

En el ejemplo siguiente se muestra cómo agrupar elementos de origen mediante algo distinto a una propiedad del objeto para la clave de grupo. En este ejemplo, la clave es la primera letra del apellido del alumno.

Pegue el método siguiente en la clase `StudentClass`. Cambie la instrucción de llamada en el método `Main` por `sc.GroupBySubstring()`.

```
public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
        }
    }
}
/* Output:
    Group by something other than a property of the object:
    Key: A
            Adams, Terry
    Key: F
            Fakhouri, Fadi
            Feng, Hanying
    Key: G
            Garcia, Cesar
            Garcia, Debra
            Garcia, Hugo
    Key: M
            Mortensen, Sven
    Key: O
            O'Donnell, Claire
            Omelchenko, Svetlana
    Key: T
            Tucker, Lance
            Tucker, Michael
    Key: Z
            Zabokritski, Eugene
*/
```

Ejemplo

En el ejemplo siguiente se muestra cómo agrupar elementos de origen mediante un intervalo numérico como la clave de grupo. Después, la consulta proyecta los resultados en un tipo anónimo que solo contiene el nombre, los apellidos y el intervalo de percentil al que pertenece el alumno. Se usa un tipo anónimo porque no es necesario usar el objeto `Student` completo para mostrar los resultados. `GetPercentile` es una función del asistente que calcula un percentil basado en la puntuación media del alumno. El método devuelve un entero entre 0 y 10.

```
//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}
```

Pegue el método siguiente en la clase `StudentClass`. Cambie la instrucción de llamada en el método `Main` por `sc.GroupByRange()`.

```
public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
        }
    }
}
/* Output:
    Group by numeric range and project into a new anonymous type:
    Key: 60
            Garcia, Debra
    Key: 70
            O'Donnell, Claire
    Key: 80
            Adams, Terry
            Feng, Hanying
            Garcia, Cesar
            Garcia, Hugo
            Mortensen, Sven
            Omelchenko, Svetlana
            Tucker, Lance
            Zabokritski, Eugene
    Key: 90
            Fakhouri, Fadi
            Tucker, Michael
*/
```

Ejemplo

En el ejemplo siguiente se muestra cómo agrupar elementos de origen usando una expresión de comparación booleana. En este ejemplo, la expresión booleana comprueba si la puntuación media del examen de un alumno es mayor de 75. Como en los ejemplos anteriores, los resultados se proyectan en un tipo anónimo porque el elemento de origen completo no es necesario. Tenga en cuenta que las propiedades del tipo anónimo se convierten en propiedades en el miembro `Key` y puede tener acceso a ellas por nombre cuando se ejecuta la consulta.

Pegue el método siguiente en la clase `StudentClass`. Cambie la instrucción de llamada en el método `Main` por `sc.GroupByBoolean()`.

```
public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                               group new { student.FirstName, student.LastName }
                                    by student.ExamScores.Average() > 75 into studentGroup
                               select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
/* Output:
    Group by a Boolean into two groups with string keys
    "True" and "False" and project into a new anonymous type:
    Key: True
            Terry Adams
            Fadi Fakhouri
            Hanying Feng
            Cesar Garcia
            Hugo Garcia
            Sven Mortensen
            Svetlana Omelchenko
            Lance Tucker
            Michael Tucker
            Eugene Zabokritski
    Key: False
            Debra Garcia
            Claire O'Donnell
*/
```

Ejemplo

En el ejemplo siguiente se muestra cómo usar un tipo anónimo para encapsular una clave que contiene varios valores. En este ejemplo, el primer valor de clave es la primera letra del apellido del alumno. El segundo valor de clave es un valor booleano que especifica si el alumno obtuvo una nota superior a 85 en el primer examen. Los grupos se pueden ordenar por cualquier propiedad de la clave.

Pegue el método siguiente en la clase `StudentClass`. Cambie la instrucción de llamada en el método `Main` por `sc.GroupByCompositeKey()`.

```
public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"\t{item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
    Group and order by a compound key:
    Name starts with A who scored more than 85
            Terry Adams
    Name starts with F who scored more than 85
            Fadi Fakhouri
            Hanying Feng
    Name starts with G who scored more than 85
            Cesar Garcia
            Hugo Garcia
    Name starts with G who scored less than 85
            Debra Garcia
    Name starts with M who scored more than 85
            Sven Mortensen
    Name starts with O who scored less than 85
            Claire O'Donnell
    Name starts with O who scored more than 85
            Svetlana Omelchenko
    Name starts with T who scored less than 85
            Lance Tucker
    Name starts with T who scored more than 85
            Michael Tucker
    Name starts with Z who scored more than 85
            Eugene Zabokritski
*/
```

Vea también

- GroupBy
- IGrouping<TKey,TElement>
- Language-Integrated Query (LINQ)
- group (cláusula)
- Tipos anónimos
- Realizar una subconsulta en una operación de agrupación
- Crear un grupo anidado
- Agrupación de datos

Crear un grupo anidado

18/03/2020 • 2 minutes to read • Edit Online

En el ejemplo siguiente se muestra cómo crear grupos anidados en una expresión de consulta LINQ. Cada grupo creado a partir del nivel académico o del año de los estudiantes se subdivide en grupos según sus nombres.

Ejemplo

> **NOTE**
>
> Este ejemplo contiene referencias a objetos que se definen en el código de ejemplo de Query a collection of objects (Consultar una colección de objetos).

```
public void QueryNestedGroups()
{
    var queryNestedGroups =
        from student in students
        group student by student.Year into newGroup1
        from newGroup2 in
            (from student in newGroup1
             group student by student.LastName)
        group newGroup2 by newGroup1.Key;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine($"\t\t{innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}
/*
 Output:
DataClass.Student Level = SecondYear
        Names that begin with: Adams
                Adams Terry
        Names that begin with: Garcia
                Garcia Hugo
        Names that begin with: Omelchenko
                Omelchenko Svetlana
DataClass.Student Level = ThirdYear
        Names that begin with: Fakhouri
                Fakhouri Fadi
        Names that begin with: Garcia
                Garcia Debra
        Names that begin with: Tucker
                Tucker Lance
DataClass.Student Level = FirstYear
        Names that begin with: Feng
                Feng Hanying
        Names that begin with: Mortensen
                Mortensen Sven
        Names that begin with: Tucker
                Tucker Michael
DataClass.Student Level = FourthYear
        Names that begin with: Garcia
                Garcia Cesar
        Names that begin with: O'Donnell
                O'Donnell Claire
        Names that begin with: Zabokritski
                Zabokritski Eugene
 */
```

Tenga en cuenta que se necesitan tres bucles `foreach` anidados para recorrer en iteración los elementos internos de un grupo anidado.

Vea también

- Language-Integrated Query (LINQ)

Realizar una subconsulta en una operación de agrupación

18/03/2020 • 2 minutes to read • Edit Online

En este artículo se muestran dos maneras diferentes de crear una consulta que ordena los datos de origen en grupos y, luego, realiza una subconsulta en cada grupo de forma individual. La técnica básica de cada ejemplo consiste en agrupar los elementos de origen usando una *continuación* denominada `newGroup` y después generar una nueva subconsulta en `newGroup`. Esta subconsulta se ejecuta en cada uno de los nuevos grupos creados por la consulta externa. Tenga en cuenta que en este ejemplo concreto el resultado final no es un grupo, sino una secuencia plana de tipos anónimos.

Para obtener más información sobre cómo agrupar, consulte Cláusula group.

Para obtener más información sobre continuaciones, consulte into. En el ejemplo siguiente se usa una estructura de datos en memoria como origen de datos, pero se aplican los mismos principios para cualquier tipo de origen de datos LINQ.

Ejemplo

> **NOTE**
>
> Este ejemplo contiene referencias a objetos que se definen en el código de ejemplo de Query a collection of objects (Consultar una colección de objetos).

```
public void QueryMax()
{
    var queryGroupMax =
        from student in students
        group student by student.Year into studentGroup
        select new
        {
            Level = studentGroup.Key,
            HighestScore =
            (from student2 in studentGroup
             select student2.ExamScores.Average()).Max()
        };

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

La consulta del fragmento de código anterior también se puede escribir con la sintaxis de método. El siguiente fragmento de código tiene una consulta semánticamente equivalente escrita con sintaxis de método.

```
public void QueryMaxUsingMethodSyntax()
{
    var queryGroupMax = students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
        });

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
    }
}
```

Vea también

- Language-Integrated Query (LINQ)

Agrupar resultados por claves contiguas

18/03/2020 • 8 minutes to read • Edit Online

En el ejemplo siguiente se muestra cómo agrupar elementos en fragmentos que representan subsecuencias de claves contiguas. Por ejemplo, suponga que tiene la siguiente secuencia de pares clave-valor:

CLAVE	VALOR
A	We
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

Los siguientes grupos se crearán en este orden:

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

La solución se implementa como un método de extensión seguro para subprocesos que devuelve sus resultados mediante transmisión por secuencias. Es decir, genera sus grupos a medida que se desplaza por la secuencia de origen. A diferencia de los operadores `group` u `orderby`, puede comenzar a devolver grupos al llamador antes de que se haya leído toda la secuencia.

La seguridad de subprocesos se logra realizando una copia de cada grupo o fragmento a medida que se recorre en iteración la secuencia de origen, tal y como se explica en los comentarios del código fuente. Si la secuencia de origen tiene una secuencia grande de elementos contiguos, Common Language Runtime puede producir una excepción OutOfMemoryException.

Ejemplo

En el ejemplo siguiente se muestran el método de extensión y el código de cliente que lo usa:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace ChunkIt
{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of each Chunk.
            // On each pass, the iterator advances to the first element of the next "Chunk"
            // in the source sequence. This loop corresponds to the outer foreach loop that
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source iterator will churn through
                // the source sequence until it finds an element with a key that doesn't match.
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
current source element.
                current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key,
keySelector(value)));

                // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the
ChunkBy method.
                // At this point the Chunk only has the first element in its source sequence. The remaining
elements will be
                // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
more info.
                yield return current;

                // Check to see whether (a) the chunk has made a copy of all its source elements or
                // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
                // foreach loop to iterate the chunk items, and that loop ran to completion,
                // then the Chunk.GetEnumerator method will already have made
                // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
                // enumerate all elements in the chunk, we need to do it here to avoid corrupting the iterator
                // for clients that may be calling us on a separate thread.
                if (current.CopyAllChunkElements() == noMoreSourceElements)
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
        // has a key and a list of ChunkItem objects, which are copies of the elements in the source sequence.
        class Chunk<TKey, TSource> : IGrouping<TKey, TSource>
        {
            // INVARIANT: DoneCopyingChunk == true ||
            //   (predicate != null && predicate(enumerator.Current) && current.Value == enumerator.Current)
```

```
            // A Chunk has a linked list of ChunkItems, which represent the elements in the current chunk. Each
ChunkItem
            // has a reference to the next ChunkItem in the list.
            class ChunkItem
            {
                public ChunkItem(TSource value)
                {
                    Value = value;
                }
                public readonly TSource Value;
                public ChunkItem Next = null;
            }

            // The value that is used to determine matching elements
            private readonly TKey key;

            // Stores a reference to the enumerator for the source sequence
            private IEnumerator<TSource> enumerator;

            // A reference to the predicate that is used to compare keys.
            private Func<TSource, bool> predicate;

            // Stores the contents of the first source element that
            // belongs with this chunk.
            private readonly ChunkItem head;

            // End of the list. It is repositioned each time a new
            // ChunkItem is added.
            private ChunkItem tail;

            // Flag to indicate the source iterator has reached the end of the source sequence.
            internal bool isLastSourceElement = false;

            // Private object for thread syncronization
            private object m_Lock;

            // REQUIRES: enumerator != null && predicate != null
            public Chunk(TKey key, IEnumerator<TSource> enumerator, Func<TSource, bool> predicate)
            {
                this.key = key;
                this.enumerator = enumerator;
                this.predicate = predicate;

                // A Chunk always contains at least one element.
                head = new ChunkItem(enumerator.Current);

                // The end and beginning are the same until the list contains > 1 elements.
                tail = head;

                m_Lock = new object();
            }

            // Indicates that all chunk elements have been copied to the list of ChunkItems,
            // and the source enumerator is either at the end, or else on an element with a new key.
            // the tail of the linked list is set to null in the CopyNextChunkElement method if the
            // key of the next element does not match the current chunk's key, or there are no more elements in
the source.
            private bool DoneCopyingChunk => tail == null;

            // Adds one ChunkItem to the current group
            // REQUIRES: !DoneCopyingChunk && lock(this)
            private void CopyNextChunkElement()
            {
                // Try to advance the iterator on the source sequence.
                // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
                isLastSourceElement = !enumerator.MoveNext();

                // If we are (a) at the end of the source, or (b) at the end of the current chunk
                // then null out the enumerator and predicate for reuse with the next chunk
```

```
        // then null out the enumerator and predicate for reuse with the next chunk.
        if (isLastSourceElement || !predicate(enumerator.Current))
        {
            enumerator = null;
            predicate = null;
        }
        else
        {
            tail.Next = new ChunkItem(enumerator.Current);
        }

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail.Next;
    }

    // Called after the end of the last chunk was reached. It first checks whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    public TKey Key => key;

    // Invoked by the inner foreach loop. This method stays just one step ahead
    // of the client requests. It adds the next element of the chunk only after
    // the clients requests the last element in the list so far.
    public IEnumerator<TSource> GetEnumerator()
    {
        //Specify the initial element to enumerate.
        ChunkItem current = head;

        // There should always be at least one ChunkItem in a Chunk.
        while (current != null)
        {
            // Yield the current item in the list.
            yield return current.Value;

            // Copy the next item from the source sequence,
            // if we are at the end of our local list.
            lock (m_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
```

```
            }

            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();
        }
    }

    // A simple named type is used for easier viewing in the debugger. Anonymous types
    // work just as well with the ChunkBy operator.
    public class KeyValPair
    {
        public string Key { get; set; }
        public string Value { get; set; }
    }

    class Program
    {
        // The source sequence.
        public static IEnumerable<KeyValPair> list;

        // Query variable declared as class member to be available
        // on different threads.
        static IEnumerable<IGrouping<string, KeyValPair>> query;

        static void Main(string[] args)
        {
            // Initialize the source sequence with an array initializer.
            list = new[]
            {
                new KeyValPair{ Key = "A", Value = "We" },
                new KeyValPair{ Key = "A", Value = "think" },
                new KeyValPair{ Key = "A", Value = "that" },
                new KeyValPair{ Key = "B", Value = "Linq" },
                new KeyValPair{ Key = "C", Value = "is" },
                new KeyValPair{ Key = "A", Value = "really" },
                new KeyValPair{ Key = "B", Value = "cool" },
                new KeyValPair{ Key = "B", Value = "!" }
            };

            // Create the query by using our user-defined query operator.
            query = list.ChunkBy(p => p.Key);

            // ChunkBy returns IGrouping objects, therefore a nested
            // foreach loop is required to access the elements in each "chunk".
            foreach (var item in query)
            {
                Console.WriteLine($"Group key = {item.Key}");
                foreach (var inner in item)
                {
                    Console.WriteLine($"\t{inner.Value}");
                }
            }

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }
    }
}
```

Para usar el método de extensión en el proyecto, copie la clase estática `MyExtensions` en un archivo de código fuente nuevo o ya existente y, si es necesario, agregue una directiva `using` para el espacio de nombres donde se encuentra.

Vea también

- Language-Integrated Query (LINQ)

Especificar dinámicamente filtros con predicado en tiempo de ejecución

18/03/2020 • 4 minutes to read • Edit Online

En algunos casos, no se conoce cuántos predicados hay que aplicar a los elementos de origen de la cláusula `where` hasta el tiempo de ejecución. Una forma de especificar dinámicamente varios filtros con predicado es usar el método Contains, como se muestra en el ejemplo siguiente. El ejemplo se construye de dos maneras. En primer lugar, el proyecto se ejecuta al filtrar por los valores proporcionados en el programa. Luego se vuelve a ejecutar mediante la entrada proporcionada en tiempo de ejecución.

Para filtrar mediante el método Contains

1. Abra una nueva aplicación de consola y denomínela `PredicateFilters`.
2. Copie la clase `StudentClass` desde Consultar una colección de objetos y péguela en el espacio de nombres `PredicateFilters` debajo de la clase `Program`. `StudentClass` proporciona una lista de objetos `Student`.
3. Comente el método `Main` de `StudentClass`.
4. Sustituya la clase `Program` por el código siguiente:

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. Agregue la siguiente línea al método `Main` de la clase `DynamicPredicates`, debajo de la declaración de `ids`.

```
QueryById(ids);
```

6. Ejecute el proyecto.
7. El resultado siguiente se muestra en una ventana de la consola:

Garcia: 114

O'Donnell: 112

Omelchenko: 111

8. El siguiente paso es volver a ejecutar el proyecto, esta vez mediante la entrada especificada en tiempo de ejecución en lugar de la matriz `ids`. Cambie `QueryByID(ids)` por `QueryByID(args)` en el método `Main`.
9. Ejecute el proyecto con los argumentos de la línea de comandos `122 117 120 115`. Una vez ejecutado el proyecto, esos valores se convierten en elementos de `args`, el parámetro del método `Main`.
10. El resultado siguiente se muestra en una ventana de la consola:

    Adams: 120

    Feng: 117

    Garcia: 115

    Tucker: 122

Para filtrar mediante una instrucción switch

1. Puede usar una instrucción `switch` para seleccionar entre consultas alternativas predeterminadas. En el ejemplo siguiente, `studentQuery` usa otra cláusula `where` en función de qué curso, o año, se especifique en tiempo de ejecución.
2. Copie el método siguiente y péguelo en la clase `DynamicPredicates`.

```
// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;

        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}
```

3. En el método `Main`, sustituya la llamada a `QueryByID` por la siguiente llamada, que envía el primer elemento de la matriz `args` como su argumento: `QueryByYear(args[0])`.

4. Ejecute el proyecto con un argumento de la línea de comandos de un valor entero entre 1 y 4.

Vea también

- Language-Integrated Query (LINQ)
- where (cláusula)

Realizar combinaciones internas

18/03/2020 • 14 minutes to read • Edit Online

En términos de la base de datos relacional, una *combinación interna* genera un conjunto de resultados en el que cada elemento de la primera colección aparece una vez para cada elemento coincidente en la segunda colección. Si un elemento de la primera colección no tiene ningún elemento coincidente, no aparece en el conjunto de resultados. El método Join, que se llama mediante la cláusula `join` de C#, implementa una combinación interna.

En este artículo se muestra cómo realizar cuatro variaciones de una combinación interna:

- Una combinación interna simple que correlaciona elementos de dos orígenes de datos según una clave simple.
- Una combinación interna que correlaciona elementos de dos orígenes de datos según una clave *compuesta*. Una clave compuesta, que es una clave formada por más de un valor, permite correlacionar elementos en función de más de una propiedad.
- Una *combinación múltiple* en la que las sucesivas operaciones de combinación se anexan entre sí.
- Una combinación interna que se implementa mediante una combinación agrupada.

Ejemplo: Combinación de clave simple

En el ejemplo siguiente se crean dos colecciones que contienen objetos de dos tipos definidos por el usuario, `Person` y `Pet`. La consulta usa la cláusula `join` de C# para emparejar objetos `Person` con objetos `Pet` cuyo `Owner` sea ese `Person`. La cláusula `select` de C# define el aspecto que tendrán los objetos resultantes. En este ejemplo, los objetos resultantes son tipos anónimos que consisten en el nombre de propietario y el nombre de mascota.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
                join pet in pets on person equals pet.Owner
                select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"\"{ownerAndPet.PetName}\" is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui
```

Tenga en cuenta que el objeto `Person` cuyo `LastName` es "Huff" no aparece en el conjunto de resultados porque no hay ningún objeto `Pet` que tenga `Pet.Owner` igual que `Person`.

Ejemplo: Combinación de clave compuesta

En lugar de correlacionar elementos en función de una sola propiedad, puede usar una clave compuesta para comparar elementos según varias propiedades. Para ello, especifique la función del selector de claves de cada colección para que devuelva un tipo anónimo que conste de las propiedades que quiere comparar. Si etiqueta las propiedades, deben tener la misma etiqueta de tipo anónimo en cada clave. Las propiedades también deben aparecer en el mismo orden.

En el ejemplo siguiente se usa una lista de objetos `Employee` y una lista de objetos `Student` para determinar qué empleados son también alumnos. Estos dos tipos tienen una propiedad `FirstName` y una propiedad `LastName` de tipo String. Las funciones que crean las claves de combinación de los elementos de cada lista devuelven un tipo anónimo formado por las propiedades `FirstName` y `LastName` de cada elemento. La operación de combinación compara la igualdad de estas claves compuestas y devuelve pares de objetos de cada lista en los que el nombre y el apellido coinciden.

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
         new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
         new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
         new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // Join the two data sources based on a composite key consisting of first and last name,
    // to determine which employees are also students.
    IEnumerable<string> query = from employee in employees
                                join student in students
                                on new { employee.FirstName, employee.LastName }
                                equals new { student.FirstName, student.LastName }
                                select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

Ejemplo: Combinación múltiple

Se puede anexar cualquier número de operaciones de combinación entre sí para realizar una combinación múltiple. Cada cláusula `join` de C# correlaciona un origen de datos especificado con los resultados de la combinación anterior.

En el ejemplo siguiente se crean tres colecciones: una lista de objetos `Person`, una lista de objetos `Cat` y una lista de objetos `Dog`.

La primera cláusula `join` de C# empareja personas y gatos según un objeto `Person` que coincida con `Cat.Owner`. Devuelve una secuencia de tipos anónimos que contienen el objeto `Person` y `Cat.Name`.

La segunda cláusula `join` de C# correlaciona los tipos anónimos devueltos por la primera combinación con objetos `Dog` de la lista de perros proporcionada, según una clave compuesta formada por la propiedad `Owner` de tipo `Person` y la primera letra del nombre del animal. Devuelve una secuencia de tipos anónimos que contienen las propiedades `Cat.Name` y `Dog.Name` de cada par coincidente. Como se trata de una combinación interna, solo se devuelven los objetos del primer origen de datos que tienen una correspondencia en el segundo origen de datos.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
                join cat in cats on person equals cat.Owner
```

```
                join dog in dogs on
                new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
                select new { CatName = cat.Name, DogName = dog.Name };

    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat \"{obj.CatName}\" shares a house, and the first letter of their name, with \"
{obj.DogName}\".");
    }
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".
```

Ejemplo: Combinación interna mediante combinación agrupada

El ejemplo siguiente muestra cómo implementar una combinación interna mediante una combinación agrupada.

En `query1`, la lista de objetos `Person` forma una combinación agrupada con la lista de objetos `Pet` según el `Person` que coincide con la propiedad `Pet.Owner`. La combinación agrupada crea una colección de grupos intermedios, donde cada grupo consta de un objeto `Person` y una secuencia de objetos `Pet` coincidentes.

Al agregar una segunda cláusula `from` a la consulta, esta secuencia de secuencias se combina (se acopla) en una secuencia mayor. El tipo de los elementos de la secuencia final se especifica con la cláusula `select`. En este ejemplo, ese tipo es un tipo anónimo que consta de las propiedades `Person.FirstName` y `Pet.Name` de cada par coincidente.

El resultado de `query1` es equivalente al conjunto de resultados que se habría obtenido con la cláusula `join` sin la cláusula `into` para realizar una combinación interna. La variable `query2` muestra esta consulta equivalente.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
```

```
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people
                 join pet in pets on person equals pet.Owner into gj
                 from subpet in gj
                 select new { OwnerName = person.FirstName, PetName = subpet.Name };

    Console.WriteLine("Inner join using GroupJoin():");
    foreach (var v in query1)
    {
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
    }

    var query2 = from person in people
                 join pet in pets on person equals pet.Owner
                 select new { OwnerName = person.FirstName, PetName = pet.Name };

    Console.WriteLine("\nThe equivalent operation using Join():");
    foreach (var v in query2)
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
```

Vea también

- Join
- GroupJoin
- Realizar combinaciones agrupadas
- Realizar operaciones de combinación externa izquierda
- Tipos anónimos

Realizar combinaciones agrupadas

04/05/2020 • 7 minutes to read • Edit Online

La combinación agrupada resulta útil para generar estructuras de datos jerárquicas. Empareja cada elemento de la primera colección con un conjunto de elementos correlacionados de la segunda colección.

Por ejemplo, una clase o una tabla de base de datos relacional denominada `Student` podría contener dos campos: `Id` y `Name`. Una segunda clase o tabla de base de datos relacional denominada `Course` podría contener dos campos: `StudentId` y `CourseTitle`. Una combinación agrupada de estos dos orígenes de datos, basada en la combinación de `Student.Id` y `Course.StudentId`, agruparía cada `Student` con una colección de objetos `Course` (que podrían estar vacíos).

> **NOTE**
>
> Cada elemento de la primera colección aparece en el conjunto de resultados de una combinación agrupada, independientemente de si se encuentran elementos correlacionados en la segunda colección. En el caso de que no se encuentren elementos correlacionados, la secuencia de elementos correlacionados para ese elemento estaría vacía. Por consiguiente, el selector de resultados tiene acceso a cada uno de los elementos de la primera colección. Esto difiere del selector de resultados en una combinación no agrupada, que no puede acceder a los elementos de la primera colección que no tienen ninguna coincidencia en la segunda colección.

> **WARNING**
>
> Enumerable.GroupJoin no tiene ningún equivalente directo en términos de base de datos relacional tradicional. Sin embargo, este método implementa un superconjunto de combinaciones internas y combinaciones externas izquierdas. Ambas operaciones se pueden escribir en términos de una combinación agrupada. Para obtener más información, consulte Operaciones de combinación y Entity Framework Core, GroupJoin.

En el primer ejemplo de este artículo se muestra cómo realizar una combinación agrupada. En el segundo ejemplo se muestra cómo usar una combinación agrupada para crear elementos XML.

Ejemplo: Combinación agrupada

En el ejemplo siguiente se realiza una combinación agrupada de objetos de tipo `Person` y `Pet` basada en la coincidencia de `Person` con la propiedad `Pet.Owner`. A diferencia de una combinación no agrupada, que generaría un par de elementos para cada coincidencia, la combinación agrupada produce un único objeto resultante para cada elemento de la primera colección, que en este ejemplo es un objeto `Person`. Los elementos correspondientes de la segunda colección, que en este ejemplo son objetos `Pet`, se agrupan en una colección. Por último, la función de selector de resultados crea un tipo anónimo para cada coincidencia formada por `Person.FirstName` y una colección de objetos `Pet`.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:
```

Ejemplo: Combinación agrupada para crear XML

Las combinaciones agrupadas resultan ideales para crear XML con LINQ to XML. El siguiente ejemplo es similar al anterior, pero en lugar de crear tipos anónimos, la función de selector de resultados crea elementos XML que

representan los objetos combinados.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));

    Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>
```

Vea también

- Join
- GroupJoin
- Realizar combinaciones internas
- Realizar operaciones de combinación externa izquierda
- Tipos anónimos (Guía de programación de C#).

Realizar operaciones de combinación externa izquierda

18/03/2020 • 3 minutes to read • Edit Online

Una combinación externa izquierda es una combinación en la que se devuelve cada elemento de la primera colección, independientemente de si tiene elementos correlacionados en la segunda colección. Puede usar LINQ para realizar una combinación externa izquierda llamando al método DefaultIfEmpty en los resultados de una combinación agrupada.

Ejemplo

En el ejemplo siguiente se muestra cómo usar el método DefaultIfEmpty en los resultados de una combinación agrupada para realizar una combinación externa izquierda.

El primer paso para generar una combinación externa izquierda de dos colecciones consiste en realizar una combinación interna usando una combinación agrupada. (Vea Realizar combinaciones internas para obtener una explicación de este proceso). En este ejemplo, la lista de objetos de `Person` está unida a la lista de objetos `Pet` basándose en un objeto `Person` que coincide con `Pet.Owner`.

El segundo paso consiste en incluir cada elemento de la primera colección (izquierda) en el conjunto de resultados, incluso cuando no haya coincidencias en la colección derecha. Esto se realiza llamando a DefaultIfEmpty en cada secuencia de elementos coincidentes de la combinación agrupada. En este ejemplo, se llama a DefaultIfEmpty en cada secuencia de objetos `Pet` coincidentes. El método devuelve una colección que contiene un único, valor predeterminado si la secuencia de objetos `Pet` coincidentes está vacía para cualquier objeto `Person`, con lo que cada objeto `Person` se representa en la colección de resultados.

> **NOTE**
>
> El valor predeterminado para un tipo de referencia es `null`; por consiguiente, el ejemplo busca una referencia NULL antes de tener acceso a cada elemento de cada colección de `Pet`.

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj.DefaultIfEmpty()
                select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName+":",-15}{v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:        Daisy
// Terry:         Barley
// Terry:         Boots
// Terry:         Blue Moon
// Charlotte:     Whiskers
// Arlene:
```

Vea también

- Join
- GroupJoin
- Realizar combinaciones internas
- Realizar combinaciones agrupadas
- Tipos anónimos (Guía de programación de C#).

Ordenar los resultados de una cláusula join

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo ordenar los resultados de una operación de combinación. Observe que la ordenación se realiza después de la combinación. Aunque puede usar una cláusula `orderby` con una o varias de las secuencias de origen antes de la combinación, normalmente no se recomienda. Algunos proveedores LINQ podrían no conservar esa ordenación después de la combinación.

Ejemplo

Esta consulta crea una combinación agrupada y luego ordena los grupos según el elemento categoría, que todavía está en el ámbito. Dentro del inicializador de tipo anónimo, una subconsulta ordena todos los elementos coincidentes de la secuencia de productos.

```
    class HowToOrderJoins
    {
        #region Data
        class Product
        {
            public string Name { get; set; }
            public int CategoryID { get; set; }
        }

        class Category
        {
            public string Name { get; set; }
            public int ID { get; set; }
        }

        // Specify the first data source.
        List<Category> categories = new List<Category>()
 {
    new Category(){Name="Beverages", ID=001},
    new Category(){ Name="Condiments", ID=002},
    new Category(){ Name="Vegetables", ID=003},
    new Category() {  Name="Grains", ID=004},
    new Category() {  Name="Fruit", ID=005}
 };

        // Specify the second data source.
        List<Product> products = new List<Product>()
{
   new Product{Name="Cola",  CategoryID=001},
   new Product{Name="Tea",  CategoryID=001},
   new Product{Name="Mustard", CategoryID=002},
   new Product{Name="Pickles", CategoryID=002},
   new Product{Name="Carrots", CategoryID=003},
   new Product{Name="Bok Choy", CategoryID=003},
   new Product{Name="Peaches", CategoryID=005},
   new Product{Name="Melons", CategoryID=005},
 };
        #endregion
        static void Main()
        {
            HowToOrderJoins app = new HowToOrderJoins();
            app.OrderJoin1();

            // Keep console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
```

```
            Console.ReadKey();
        }

        void OrderJoin1()
        {
            var groupJoinQuery2 =
                from category in categories
                join prod in products on category.ID equals prod.CategoryID into prodGroup
                orderby category.Name
                select new
                {
                    Category = category.Name,
                    Products = from prod2 in prodGroup
                               orderby prod2.Name
                               select prod2
                };

            foreach (var productGroup in groupJoinQuery2)
            {
                Console.WriteLine(productGroup.Category);
                foreach (var prodItem in productGroup.Products)
                {
                    Console.WriteLine($"  {prodItem.Name,-10} {prodItem.CategoryID}");
                }
            }
        }
        /* Output:
            Beverages
              Cola       1
              Tea        1
            Condiments
              Mustard    2
              Pickles    2
            Fruit
              Melons     5
              Peaches    5
            Grains
            Vegetables
              Bok Choy   3
              Carrots    3
         */
    }
```

Vea también

- Language-Integrated Query (LINQ)
- orderby (cláusula)
- join (cláusula)

Realizar una unión usando claves compuestas

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo realizar operaciones de combinación en las que se quiere usar más de una clave para definir a una coincidencia. Esto se logra mediante una clave compuesta. Una clave compuesta se crea como un tipo anónimo o un nombre con tipo con los valores que se quieren comparar. Si la variable de consulta se pasará a través de límites de método, use un tipo con nombre que invalida Equals y GetHashCode para la clave. Los nombres de las propiedades y el orden en que aparecen deben ser idénticos en cada clave.

Ejemplo

En el ejemplo siguiente se muestra cómo usar una clave compuesta para combinar datos de tres tablas:

```
var query = from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID} into details
        from d in details
        select new {o.OrderID, p.ProductID, d.UnitPrice};
```

La inferencia de tipos en las claves compuestas depende de los nombres de las propiedades de las claves y el orden en que aparecen. Si las propiedades de las secuencias de origen no tienen los mismos nombres, deberá asignar nombres nuevos en las claves. Por ejemplo, si la tabla `Orders` y la tabla `OrderDetails` usan nombres diferentes para las columnas, se podrían crear claves compuestas asignando nombres idénticos a los tipos anónimos:

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
    new {Name = d.CustName, ID = d.CustID }
```

Las claves compuestas también se pueden usar en una cláusula `group`.

Vea también

- Language-Integrated Query (LINQ)
- join (cláusula)
- group (cláusula)

Realizar operaciones de combinación personalizadas

18/03/2020 • 7 minutes to read • Edit Online

En este ejemplo se muestra cómo realizar operaciones de combinación que no son posibles con la cláusula `join`. En una expresión de consulta, la cláusula `join` se limita a combinaciones de igualdad (y se optimiza para estas), que son con diferencia el tipo más común de operación de combinación. Al realizar una combinación de igualdad, probablemente obtendrá siempre el mejor rendimiento con la cláusula `join`.

En cambio, la cláusula `join` no se puede usar en los siguientes casos:

- Cuando la combinación se basa en una expresión de desigualdad (una combinación que no es de igualdad).
- Cuando la combinación se basa en más de una expresión de igualdad o desigualdad.
- Cuando tenga que introducir una variable de rango temporal para la secuencia de la derecha (interior) antes de la operación de combinación.

Para realizar combinaciones que no son de igualdad, puede usar varias cláusulas `from` para presentar cada origen de datos de forma independiente. Después, aplique una expresión de predicado de una cláusula `where` a la variable de rango para cada origen. La expresión también puede adoptar la forma de una llamada de método.

> **NOTE**
>
> No confunda este tipo de operación de combinación personalizada con el uso de varias cláusulas `from` para acceder a colecciones internas. Para obtener más información, vea join (Cláusula, Referencia de C#).

Ejemplo

En el primer método del siguiente ejemplo se muestra una combinación cruzada sencilla. Las combinaciones cruzadas se deben usar con precaución porque pueden generar conjuntos de resultados muy grandes. En cambio, pueden resultar útiles en algunos escenarios para crear secuencias de origen en las que se ejecutan consultas adicionales.

El segundo método genera una secuencia de todos los productos cuyo identificador de categoría aparece en la lista de categorías en el lado izquierdo. Observe el uso de la cláusula `let` y el método `Contains` para crear una matriz temporal. También se puede crear la matriz antes de la consulta y eliminar la primera cláusula `from`.

```
class CustomJoins
{

    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
```

```
        // Specify the first data source.
        List<Category> categories = new List<Category>()
        {
            new Category(){Name="Beverages", ID=001},
            new Category(){ Name="Condiments", ID=002},
            new Category(){ Name="Vegetables", ID=003},
        };

        // Specify the second data source.
        List<Product> products = new List<Product>()
        {
            new Product{Name="Tea",  CategoryID=001},
            new Product{Name="Mustard", CategoryID=002},
            new Product{Name="Pickles", CategoryID=002},
            new Product{Name="Carrots", CategoryID=003},
            new Product{Name="Bok Choy", CategoryID=003},
            new Product{Name="Peaches", CategoryID=005},
            new Product{Name="Melons", CategoryID=005},
            new Product{Name="Ice Cream", CategoryID=007},
            new Product{Name="Mackerel", CategoryID=012},
        };
        #endregion

        static void Main()
        {
            CustomJoins app = new CustomJoins();
            app.CrossJoin();
            app.NonEquijoin();

            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }

        void CrossJoin()
        {
            var crossJoinQuery =
                from c in categories
                from p in products
                select new { c.ID, p.Name };

            Console.WriteLine("Cross Join Query:");
            foreach (var v in crossJoinQuery)
            {
                Console.WriteLine($"{v.ID,-5}{v.Name}");
            }
        }

        void NonEquijoin()
        {
            var nonEquijoinQuery =
                from p in products
                let catIds = from c in categories
                             select c.ID
                where catIds.Contains(p.CategoryID) == true
                select new { Product = p.Name, CategoryID = p.CategoryID };

            Console.WriteLine("Non-equijoin query:");
            foreach (var v in nonEquijoinQuery)
            {
                Console.WriteLine($"{v.CategoryID,-5}{v.Product}");
            }
        }
    }
    /* Output:
Cross Join Query:
1    Tea
1    Mustard
1    Pickles
1    Carrots
1    Bok Choy
```

```
1    Bok Choy
1    Peaches
1    Melons
1    Ice Cream
1    Mackerel
2    Tea
2    Mustard
2    Pickles
2    Carrots
2    Bok Choy
2    Peaches
2    Melons
2    Ice Cream
2    Mackerel
3    Tea
3    Mustard
3    Pickles
3    Carrots
3    Bok Choy
3    Peaches
3    Melons
3    Ice Cream
3    Mackerel
Non-equijoin query:
1    Tea
2    Mustard
2    Pickles
3    Carrots
3    Bok Choy
Press any key to exit.
     */
```

Ejemplo

En el ejemplo siguiente, la consulta debe combinar dos secuencias basándose en las claves coincidentes que, en el caso de la secuencia interna (lado derecho), no se pueden obtener antes de la propia cláusula de combinación. Si esta combinación se realizara con una cláusula `join`, se tendría que llamar al método `Split` para cada elemento. El uso de varias cláusulas `from` permite a la consulta evitar la sobrecarga que supone la llamada al método repetida. En cambio, puesto que `join` está optimizada, en este caso particular puede que siga resultando más rápido que usar varias cláusulas `from`. Los resultados variarán dependiendo principalmente de cuántos recursos requiera la llamada de método.

```
class MergeTwoCSVFiles
{
    static void Main()
    {
        // See section Compiling the Code for information about the data files.
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Merge the data sources using a named type.
        // You could use var instead of an explicit type for the query.
        IEnumerable<Student> queryNamesScores =
            // Split each line in the data files into an array of strings.
            from name in names
            let x = name.Split(',')
            from score in scores
            let s = score.Split(',')
            // Look for matching IDs from the two data files.
            where x[2] == s[0]
            // If the IDs match, build a Student object.
            select new Student()
            {
                FirstName = x[0],
                LastName = x[1],
```

```
                LastName = x[1],
                ID = Convert.ToInt32(x[2]),
                ExamScores = (from scoreAsText in s.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries
        List<Student> students = queryNamesScores.ToList();

        foreach (var student in students)
        {
            Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
{student.ExamScores.Average()}.");
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

Vea también

- Language-Integrated Query (LINQ)
- join (cláusula)
- Ordenar los resultados de una cláusula join

Controlar valores nulos en expresiones de consulta

02/04/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo controlar los posibles valores nulos en colecciones de origen. Una colección de objetos como IEnumerable<T> puede contener elementos cuyo valor es NULL. Si una colección de origen es nula o contiene un elemento cuyo valor es NULL, y la consulta no controla los valores NULL, se iniciará una NullReferenceException cuando se ejecute la consulta.

Ejemplo

Se pueden codificar de forma defensiva para evitar una excepción de referencia nula, tal y como se muestra en el ejemplo siguiente:

```
var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals
        p?.CategoryID
    select new { Category = c.Name, Name = p.Name };
```

En el ejemplo anterior, la cláusula `where` filtra todos los elementos nulos de la secuencia de categorías. Esta técnica es independiente de la comprobación de null en la cláusula join. La expresión condicional con null de este ejemplo funciona porque `Products.CategoryID` es de tipo `int?`, que es una abreviatura de `Nullable<int>`.

Ejemplo

En una cláusula join, si solo una de las claves de comparación es de un tipo que acepta valores NULL, puede convertir la otra en un tipo que acepta valores NULL en la expresión de consulta. En el ejemplo siguiente, suponga que `EmployeeID` es una columna que contiene valores de tipo `int?`:

```
void TestMethod(Northwind db)
{
    var query =
        from o in db.Orders
        join e in db.Employees
            on o.EmployeeID equals (int?)e.EmployeeID
        select new { o.OrderID, e.FirstName };
}
```

Vea también

- Nullable<T>
- Language-Integrated Query (LINQ)
- Tipos de valores que aceptan valores NULL

Controlar excepciones en expresiones de consulta

18/03/2020 • 4 minutes to read • Edit Online

Es posible llamar a cualquier método en el contexto de una expresión de consulta. Pero se recomienda evitar llamar a cualquier método en una expresión de consulta que pueda crear un efecto secundario, como modificar el contenido del origen de datos o producir una excepción. En este ejemplo se muestra cómo evitar que se produzcan excepciones al llamar a métodos en una expresión de consulta sin infringir las instrucciones generales de .NET sobre el control de excepciones. En esas instrucciones se indica que es aceptable detectar una excepción concreta cuando se comprende por qué se produce en un contexto determinado. Para obtener más información, vea Procedimientos recomendados para excepciones.

En el último ejemplo se muestra cómo controlar los casos en los que se debe producir una excepción durante la ejecución de una consulta.

Ejemplo

En el ejemplo siguiente se muestra cómo mover código de control de excepciones fuera de una expresión de consulta. Esto solo es posible cuando el método no depende de ninguna variable local de la consulta.

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // If we get here, it is safe to proceed.
        var query = from i in dataSource
                    select i * i;

        foreach (var i in query)
            Console.WriteLine(i.ToString());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

Ejemplo

En algunos casos, la mejor respuesta a una excepción que se produce dentro de una consulta podría ser detener la ejecución de la consulta inmediatamente. En el ejemplo siguiente se muestra cómo controlar las excepciones que pueden producirse desde el cuerpo de una consulta. Supongamos que `SomeMethodThatMightThrow` puede producir una excepción que requiere que se detenga la ejecución de la consulta.

Tenga en cuenta que el bloque `try` encierra el bucle `foreach` y no la propia consulta. Esto es porque el bucle `foreach` es el punto en el que se ejecuta realmente la consulta. Para obtener más información, vea Introduction to LINQ queries (Introducción a las consultas LINQ).

```
class QueryThatThrows
{
    static void Main()
    {
        // Data source.
        string[] files = { "fileA.txt", "fileB.txt", "fileC.txt" };

        // Demonstration query that throws.
        var exceptionDemoQuery =
            from file in files
            let n = SomeMethodThatMightThrow(file)
            select n;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

Vea también

- Language-Integrated Query (LINQ)

Programación asincrónica

29/06/2020 • 20 minutes to read • Edit Online

Si tiene cualquier necesidad enlazada a E/S (por ejemplo, solicitar datos de una red, acceder a una base de datos o leer y escribir un sistema de archivos), deberá usar la programación asincrónica. También podría tener código enlazado a la CPU, como realizar un cálculo costoso, que también es un buen escenario para escribir código asincrónico.

C# tiene un modelo de programación asincrónico de nivel de lenguaje que permite escribir fácilmente código asincrónico sin tener que hacer malabares con las devoluciones de llamada o ajustarse a una biblioteca que admita la asincronía. Sigue lo que se conoce como el modelo asincrónico basado en tareas (TAP).

Información general del modelo asincrónico

El núcleo de la programación asincrónica son los objetos `Task` y `Task<T>`, que modelan las operaciones asincrónicas. Son compatibles con las palabras clave `async` y `await`. El modelo es bastante sencillo en la mayoría de los casos:

- Para el código enlazado a E/S, espera una operación que devuelva `Task` o `Task<T>` dentro de un método `async`.
- Para el código enlazado a la CPU, espera una operación que se inicia en un subproceso en segundo plano con el método Task.Run.

La palabra clave `await` es donde ocurre la magia. Genera control para el autor de la llamada del método que ha realizado `await`, y permite en última instancia una interfaz de usuario con capacidad de respuesta o un servicio flexible. Aunque existen maneras de abordar el código asincrónico diferentes de `async` y `await`, este artículo se centra en las construcciones de nivel de lenguaje.

Ejemplo enlazado a E/S: descarga de datos de un servicio web

Puede que necesite descargar algunos datos de un servicio web cuando se presione un botón, pero no quiere bloquear el subproceso de interfaz de usuario. Puede conseguirlo de la siguiente forma:

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

El código expresa la intención (descargar datos de forma asincrónica) sin verse obstaculizado en la interacción con objetos `Task`.

Ejemplo enlazado a la CPU: realizar un cálculo para un juego

Supongamos que está escribiendo un juego para móviles en el que se pueden infligir daños a muchos enemigos en la pantalla pulsando un botón. Realizar el cálculo del daño puede resultar costoso y hacerlo en el subproceso de interfaz de usuario haría que pareciera que el juego se pone en pausa mientras se lleva a cabo el cálculo.

La mejor manera de abordar esta situación consiste en iniciar un subproceso en segundo plano que realice la tarea mediante `Task.Run` y esperar su resultado mediante `await`. Esto permite que la interfaz de usuario funcione de manera fluida mientras se lleva a cabo la tarea.

```
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}

calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

Este código expresa claramente la intención del evento de clic del botón, no requiere la administración manual de un subproceso en segundo plano y lo hace en un modo sin bloqueo.

Qué sucede en segundo plano

En las operaciones asincrónicas existen numerosos aspectos dinámicos. Si siente curiosidad sobre lo que ocurre en el segundo plano de `Task` y `Task<T>`, eche un vistazo al artículo Async en profundidad para obtener más información.

En lo que respecta a C#, el compilador transforma el código en una máquina de estados que realiza el seguimiento de acciones como la retención de la ejecución cuando se alcanza `await` y la reanudación de la ejecución cuando se ha finalizado un trabajo en segundo plano.

Para los más interesados en la teoría, se trata de una implementación del modelo de promesas de asincronía.

Piezas clave que debe comprender

- El código asincrónico puede usarse para código tanto enlazado a E/S como enlazado a la CPU, pero de forma distinta en cada escenario.
- El código asincrónico usa `Task<T>` y `Task`, que son construcciones que se usan para modelar el trabajo que se realiza en segundo plano.
- La palabra clave `async` convierte un método en un método asincrónico, lo que permite usar la palabra clave `await` en su cuerpo.
- Cuando se aplica la palabra clave `await`, se suspende el método de llamada y se cede el control al autor de la llamada hasta que se completa la tarea esperada.
- `await` solo puede usarse dentro de un método asincrónico.

Reconocer el trabajo enlazado a la CPU y el enlazado a E/S

En los dos primeros ejemplos de esta guía se ha explicado cómo se puede usar `async` y `await` para trabajos enlazados a E/S y a la CPU. Resulta fundamental que pueda identificar si el trabajo que debe realizar está enlazado a E/S o a la CPU, ya que esto puede afectar en gran medida al rendimiento del código y podría dar lugar al uso inadecuado de ciertas construcciones.

A continuación, se indican dos preguntas que debe hacerse antes de escribir el código:

1. ¿Estará su código "esperando" algo, como datos de una base de datos?

Si la respuesta es "sí", su trabajo está **enlazado a E/S**.

2. ¿Realizará el código un cálculo costoso?

   Si la respuesta es "sí", su trabajo está **enlazado a la CPU**.

Si el trabajo que tiene está **enlazado a E/S**, use `async` y `await` *sin* `Task.Run`. *No debe* usar la Biblioteca TPL. Esto se explica en Async en profundidad.

Si el trabajo que tiene está **enlazado a la CPU** y le interesa la capacidad de respuesta, use `async` y `await`, pero genere el trabajo en otro subproceso *con* `Task.Run`. Si el trabajo es adecuado para la simultaneidad y el paralelismo, también debe plantearse el uso de la biblioteca TPL.

Además, siempre debe medir la ejecución del código. Por ejemplo, puede verse en una situación en la que el trabajo enlazado a la CPU no sea suficientemente costoso en comparación con la sobrecarga de cambios de contexto cuando realice multithreading. Cada opción tiene su compensación y debe elegir el equilibrio correcto para su situación.

Más ejemplos

En los ejemplos siguientes se muestran distintas maneras en las que puede escribir código asincrónico en C#. Abarcan algunos escenarios diferentes con los que puede encontrarse.

Extracción de datos de una red

Este fragmento de código descarga el HTML desde la página principal en https://dotnetfoundation.org y cuenta el número de veces que aparece la cadena ".NET" en el código HTML. Usa ASP.NET para definir un método de controlador Web API que realiza esta tarea y devuelve el número.

> **NOTE**
> Si tiene previsto realizar un análisis HTML en el código de producción, no use expresiones regulares. Use una biblioteca de análisis en su lugar.

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await _httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.NET").Count;
}
```

Este es el mismo escenario escrito para una aplicación Windows Universal, que realiza la misma tarea cuando se presiona un botón:

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask = _httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a Progress Bar.
    // This is important to do here, before the "await" call, so that the user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends SeeTheDotNets_Click, returning control to its caller.
    // This is what allows the app to be responsive and not block the UI thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

Esperar a que se completen varias tareas

Es posible que se vea en una situación en la que necesite recuperar varios fragmentos de datos al mismo tiempo. La API `Task` contiene dos métodos, Task.WhenAll y Task.WhenAny, que permiten escribir código asincrónico que realiza una espera sin bloqueo en varios trabajos en segundo plano.

En este ejemplo se muestra cómo podría captar datos `User` de un conjunto de elementos `userId`.

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

Aquí tiene otra manera de escribir lo mismo de una forma más sucinta, con LINQ:

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id));
    return await Task.WhenAll(getUserTasks);
}
```

Aunque es menos código, tenga cuidado al combinar LINQ con código asincrónico. Dado que LINQ usa la ejecución diferida, las llamadas asincrónicas no se realizarán inmediatamente, como lo hacen en un bucle `foreach`, a menos que fuerce la secuencia generada a procesar una iteración con una llamada a `.ToList()` o `.ToArray()`.

Consejos e información importante

Con la programación asincrónica, hay algunos detalles que debe tener en cuenta para evitar un comportamiento inesperado.

- Los métodos `async` **deben tener una palabra clave** `await` **en el cuerpo o nunca proporcionarán resultados**.

  Es importante que tenga esto en cuenta. Si no se usa `await` en el cuerpo de un método `async`, el compilador de C# genera una advertencia, pero el código se compila y se ejecuta como si se tratara de un método normal. Esto sería muy ineficaz, ya que la máquina de estados generada por el compilador de C# para el método asincrónico no realiza nada.

- **Debe agregar "Async" como el sufijo de todos los métodos asincrónicos que escriba.**

Se trata de la convención que se usa en .NET para distinguir más fácilmente los métodos sincrónicos de los asincrónicos. No se aplican necesariamente ciertos métodos a los que el código no llame explícitamente (como controladores de eventos o métodos de controlador web). Puesto que el código no los llama explícitamente, resulta importante explicitar sus nombres.

- `async void` **solo se debe usar para controladores de eventos.**

`async void` es la única manera de permitir a los controladores de eventos asincrónicos trabajar, ya que los eventos no tienen tipos de valor devuelto (por lo tanto, no pueden hacer uso de `Task` y `Task<T>`). Cualquier otro uso de `async void` no sigue el modelo de TAP y puede resultar difícil de usar, como:

- Las excepciones producidas en un método `async void` no se pueden detectar fuera de ese método.
- Los métodos `async void` resultan muy difíciles de probar.
- Los métodos `async void` pueden provocar efectos secundarios negativos si el autor de la llamada no espera que sean asincrónicos.

- **Tenga cuidado al usar lambdas asincrónicas en las expresiones de LINQ.**

Las expresiones lambda de LINQ usan la ejecución aplazada, lo que implica que el código podría acabar ejecutándose en un momento en que no se lo espere. La introducción de las tareas de bloqueo puede dar lugar a un interbloqueo si no se han escrito correctamente. Además, el anidamiento de código asincrónico de esta manera también puede hacer que resulte más difícil razonar sobre la ejecución del código. Async y LINQ son

eficaces, pero deben usarse conjuntamente con el mayor cuidado y claridad posible.

- **Escriba código que espere las tareas sin bloqueo.**

Bloquear el subproceso actual como un medio para esperar que se complete `Task` puede dar lugar a interbloqueos y subprocesos de contexto bloqueados, y puede requerir un control de errores más complejo. En la tabla siguiente se ofrece orientación sobre cómo abordar la espera de las tareas de una manera que no produzca un bloqueo:

USE ESTO...	EN VEZ DE ESTO...	CUANDO QUIERA HACER ESTO...
`await`	`Task.Wait` o `Task.Result`	Recuperar el resultado de una tarea en segundo plano
`await Task.WhenAny`	`Task.WaitAny`	Esperar que finalice cualquier tarea
`await Task.WhenAll`	`Task.WaitAll`	Esperar que finalicen todas las tareas
`await Task.Delay`	`Thread.Sleep`	Esperar un período de tiempo

- **Considere la posibilidad de usar `ValueTask` cuando sea posible**

La devolución de un objeto `Task` desde métodos asincrónicos puede presentar cuellos de botella de rendimiento en determinadas rutas de acceso. `Task` es un tipo de referencia, por lo que su uso implica la asignación de un objeto. En los casos en los que un método declarado con el modificador `async` devuelva un resultado en caché o se complete sincrónicamente, las asignaciones adicionales pueden suponer un costo considerable de tiempo en secciones críticas para el rendimiento del código. Esas asignaciones pueden resultar costosas si se producen en bucles ajustados. Para obtener más información, consulte Tipos de valor devueltos asincrónicos generalizados.

- **Considere la posibilidad de utilizar `ConfigureAwait(false)`**

Una pregunta habitual es "¿Cuándo debo usar el método Task.ConfigureAwait(Boolean)?". El método permite a una instancia de `Task` configurar su elemento awaiter. Este es un aspecto importante que debe tenerse en cuenta, y su configuración incorrecta podría tener implicaciones de rendimiento e incluso interbloqueos. Para obtener más información sobre `ConfigureAwait`, consulte las preguntas más frecuentes sobre ConfigureAwait.

- **Escriba código con menos estados.**

No dependa del estado de los objetos globales o la ejecución de ciertos métodos. En su lugar, dependa únicamente de los valores devueltos de los métodos. ¿Por qué?

- Le resultará más fácil razonar sobre el código.
- Le resultará más fácil probar el código.
- Resulta mucho más sencillo mezclar código asincrónico y sincrónico.
- Normalmente se pueden evitar por completo las condiciones de carrera.
- Depender de los valores devueltos facilita la coordinación de código asincrónico.
- (Extra) Funciona muy bien con la inserción de dependencias.

Un objetivo recomendado es lograr una transparencia referencial completa o casi completa en el código. Esto se traducirá en un código base extremadamente predecible, que se puede probar y es fácil de mantener.

Otros recursos

- En el artículo Async en profundidad se proporciona más información sobre cómo funcionan las tareas.
- Programación asincrónica con async y await (C#)

- Los vídeos Six Essential Tips for Async (Seis consejos esenciales para la programación asincrónica) de Lucian Wischik son un recurso fantástico para este tipo de programación.

Coincidencia de modelos

02/04/2020 • 24 minutes to read • Edit Online

Los patrones comprueban que un valor tenga una determinada *forma* y pueden *extraer* información del valor cuando tiene la forma coincidente. La coincidencia de patrones proporciona una sintaxis más concisa para los algoritmos que se usan actualmente. Ya se crean algoritmos de coincidencia de patrones mediante la sintaxis existente. Se escriben instrucciones `if` o `switch` que comprueban valores. Luego, si esas instrucciones coinciden, se extrae y se usa la información de ese valor. Los nuevos elementos de sintaxis son extensiones de instrucciones con las que ya está familiarizado: `is` y `switch`. Estas nuevas extensiones combinan la comprobación de un valor y la extracción de esa información.

En este artículo se tratará la nueva sintaxis para mostrar cómo permite escribir un código conciso y legible. La coincidencia de patrones permite expresiones donde se separan el código y los datos, a diferencia de los diseños orientados a objetos, donde los datos y los métodos que los manipulan están estrechamente unidos.

Para mostrar estas nuevas expresiones, vamos a trabajar con estructuras que representan formas geométricas mediante instrucciones de coincidencia de patrones. Probablemente esté familiarizado con la creación de jerarquías de clases y de métodos virtuales y métodos invalidados para personalizar el comportamiento de los objetos según el tipo de tiempo de ejecución del objeto.

Esas técnicas no son posibles en el caso de los datos que no están estructurados en una jerarquía de clases. Cuando los datos y los métodos están separados, se necesitan otras herramientas. Las nuevas construcciones de *coincidencia de patrones* permiten una sintaxis más limpia para examinar los datos y manipular el flujo de control basándose en cualquier condición de esos datos. Ya escribe instrucciones `if` y `switch` que comprueban el valor de una variable. Escribe instrucciones `is` que comprueban el tipo de una variable. La *coincidencia de patrones* agrega nuevas capacidades a esas instrucciones.

En este artículo se creará un método que calcula el área de distintas formas geométricas. Pero se hará sin recurrir a técnicas orientadas a objetos y sin crear una jerarquía de clases para las diferentes formas. En lugar de esto se usará la *coincidencia de patrones*. Conforme avance en este ejemplo, compare este código con cómo se estructuraría como una jerarquía de objetos. Cuando los datos que se deben consultar y manipular no son una jerarquía de clases, la coincidencia de patrones permite diseños elegantes.

En lugar de empezar con una definición de forma abstracta y agregar diferentes clases de formas concretas, se comenzará con simples definiciones solo de datos para cada una de las formas geométricas:

```
public class Square
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }
}
public class Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }
}
public struct Rectangle
{
    public double Length { get; }
    public double Height { get; }

    public Rectangle(double length, double height)
    {
        Length = length;
        Height = height;
    }
}
public class Triangle
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }
}
```

A partir de estas estructuras se va a escribir un método que calcula el área de alguna forma.

Expresión de patrón de tipo `is`

Antes de C# 7.0, había que comprobar cada tipo en una serie de instrucciones `if` e `is`:

```
public static double ComputeArea(object shape)
{
    if (shape is Square)
    {
        var s = (Square)shape;
        return s.Side * s.Side;
    }
    else if (shape is Circle)
    {
        var c = (Circle)shape;
        return c.Radius * c.Radius * Math.PI;
    }
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}
```

El código anterior es una expresión clásica del *patrón de tipo*: se prueba una variable para determinar su tipo y se realiza una acción diferente basada en ese tipo.

Este código se simplifica con extensiones de la expresión `is` para asignar una variable si la prueba se realiza correctamente:

```
public static double ComputeAreaModernIs(object shape)
{
    if (shape is Square s)
        return s.Side * s.Side;
    else if (shape is Circle c)
        return c.Radius * c.Radius * Math.PI;
    else if (shape is Rectangle r)
        return r.Height * r.Length;
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}
```

En esta versión actualizada, la expresión `is` prueba la variable y la asigna a una nueva variable del tipo correcto. Observe también que esta versión incluye el tipo `Rectangle`, que es un elemento `struct`. La nueva expresión `is` funciona con tipos de valor y con tipos de referencia.

Las reglas del lenguaje para las expresiones de coincidencia de patrones ayudan a evitar el uso indebido de los resultados de una expresión de coincidencia. En el ejemplo anterior, las variables `s`, `c` y `r` solo están en el ámbito y se asignan definitivamente cuando las expresiones de coincidencia de patrones respectivas tienen resultados `true`. Si intenta usar una de las variables en otra ubicación, el código genera errores del compilador.

Vamos a examinar detenidamente esas dos reglas, a partir del ámbito. La variable `c` está en el ámbito únicamente en la rama `else` de la primera instrucción `if`. La variable `s` está en el ámbito en el método `ComputeAreaModernIs`. Eso se debe a que cada rama de una instrucción `if` establece un ámbito independiente para las variables. Pero la propia instrucción `if` no. Eso significa que las variables declaradas en la instrucción `if` están en el mismo ámbito que la instrucción `if` (el método en este caso). Este comportamiento no es específico de la coincidencia de patrones, sino que es el definido para los ámbitos de variable y las instrucciones `if` y `else`.

Las variables `c` y `s` se asignan cuando las respectivas instrucciones `if` son true debido al mecanismo when true asignado definitivamente.

TIP

En los ejemplos de este tema se usa la construcción recomendada, donde una expresión de coincidencia de patrones `is` asigna definitivamente la variable de coincidencia en la rama `true` de la instrucción `if`. Se podría invertir la lógica al decir `if (!(shape is Square s))` y la variable `s` se asignaría definitivamente solo en la rama `false`. Aunque esto es C# válido, no se recomienda, porque es más confuso para seguir la lógica.

Estas reglas significan que es poco probable que se acceda accidentalmente al resultado de una expresión de coincidencia de patrones cuando no ha habido coincidencia de ese patrón.

Uso de instrucciones de coincidencia de patrones `switch`

Con el tiempo, es posible que tenga que admitir otros tipos de formas. A medida que crece el número de condiciones que se está probando, puede resultar pesado el uso de expresiones de coincidencia de patrones `is`. Además de necesitar instrucciones `if` en cada tipo que se quiere comprobar, las expresiones `is` solo se pueden probar si la entrada coincide con un único tipo. En este caso, las expresiones de coincidencia de patrones `switch` son una mejor opción.

La instrucción tradicional `switch` era una expresión de patrón: admitía el patrón de constante. Se podía comparar una variable con cualquier constante usada en una instrucción `case`:

```
public static string GenerateMessage(params string[] parts)
{
    switch (parts.Length)
    {
        case 0:
            return "No elements to the input";
        case 1:
            return $"One element: {parts[0]}";
        case 2:
            return $"Two elements: {parts[0]}, {parts[1]}";
        default:
            return $"Many elements. Too many to write";
    }
}
```

El único patrón admitido por la instrucción `switch` era el patrón de constante. Se limitaba aún más a tipos numéricos y al tipo `string`. Esas restricciones se han eliminado y ahora se puede escribir una instrucción `switch` con el patrón de tipos:

```
public static double ComputeAreaModernSwitch(object shape)
{
    switch (shape)
    {
        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Rectangle r:
            return r.Height * r.Length;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

La instrucción de coincidencia de patrones `switch` usa una sintaxis familiar para los desarrolladores que han empleado la instrucción de estilo C tradicional `switch`. Cada `case` se evalúa y se ejecuta el código debajo de la condición que coincide con la variable de entrada. La ejecución de código no puede "pasar explícitamente" de una expresión case a la siguiente; la sintaxis de la instrucción `case` exige que cada `case` termine con `break`, `return` o `goto`.

> **NOTE**
> Las instrucciones `goto` para saltar a otra etiqueta solo son válidas para el patrón de constante, la instrucción switch clásica.

Hay nuevas e importantes reglas que rigen la instrucción `switch`. Las restricciones respecto al tipo de la variable en la expresión `switch` se han eliminado. Se puede usar cualquier tipo, como `object` en este ejemplo. Las expresiones case ya no se limitan a valores constantes. La eliminación de esa limitación significa que la reordenación de secciones `switch` puede cambiar el comportamiento de un programa.

Cuando se limitaba a valores constantes, más de una etiqueta `case` podía coincidir con el valor de la expresión `switch`. Eso sumado a la regla de que cada sección `switch` no debe pasar explícitamente a la sección siguiente, el resultado era que las secciones `switch` se podían reorganizar en cualquier orden sin afectar al comportamiento. Ahora, con expresiones `switch` más generalizadas, el orden de cada sección importa. Las expresiones `switch` se evalúan en orden textual. La ejecución se transfiere a la primera etiqueta `switch` que coincide con la expresión `switch`.
El caso `default` solo se ejecutará si no coincide ninguna otra etiqueta case. El caso `default` se evalúa en último lugar, independientemente de su orden textual. Si no hay ningún caso `default` y ninguna de las instrucciones `case` coincide, la ejecución continúa en la instrucción siguiente a la instrucción `switch`. No se ejecuta el código de ninguna de las etiquetas `case`.

Cláusulas `when` en expresiones `case`

Puede crear casos especiales para las formas que tengan área 0 mediante una cláusula `when` en la etiqueta `case`. Un cuadrado con una longitud de lado de 0 o un círculo con un radio de 0 tiene un área 0. Esa condición se especifica mediante una cláusula `when` en la etiqueta `case`:

```
public static double ComputeArea_Version3(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

Este cambio muestra algunos puntos importantes sobre la nueva sintaxis. En primer lugar, se pueden aplicar varias etiquetas `case` a una sección `switch`. El bloque de instrucciones se ejecuta cuando cualquiera de esas etiquetas es `true`. En esta instancia, si la expresión `switch` es un círculo o un cuadrado con área 0, el método devuelve la constante 0.

Este ejemplo presenta dos variables distintas en las dos etiquetas `case` del primer bloque `switch`. Observe que las instrucciones de este bloque `switch` no usan las variables `c` (para el círculo) ni `s` (para el cuadrado). Ninguna de esas variables se ha asignado definitivamente en este bloque `switch`. Si alguno de estos casos coincide, claramente se ha asignado una de las variables. Pero no es posible saber *cuál* se ha asignado en tiempo de compilación, ya que cualquiera de los casos podría coincidir en tiempo de ejecución. Por ese motivo, la mayoría de las veces en que se usan varias etiquetas `case` para el mismo bloque, no se presenta una nueva variable en la instrucción `case` o solo se usará la variable en la cláusula `when`.

Una vez agregadas esas formas con área 0, se van a agregar un par de tipos de formas más: un rectángulo y un triángulo:

```
public static double ComputeArea_Version4(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

Este conjunto de cambios agrega etiquetas `case` para el caso degenerado y etiquetas y bloques para cada una de las nuevas formas.

Por último, puede agregar un caso `null` para garantizar que el argumento no sea `null`:

```
public static double ComputeArea_Version5(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        case null:
            throw new ArgumentNullException(paramName: nameof(shape), message: "Shape must not be null");
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

El comportamiento especial del patrón `null` es interesante porque la constante `null` del patrón no tiene un tipo, pero se puede convertir a cualquier tipo de referencia o tipo que acepte valores NULL. En lugar de convertir `null` en cualquier tipo, el lenguaje define que un valor `null` no coincidirá con ningún patrón de tipo, independientemente del tipo de tiempo de compilación de la variable. Este comportamiento hace que el nuevo patrón de tipo basado en `switch` sea coherente con la instrucción `is`: las instrucciones `is` siempre devuelven `false` cuando el valor que se está comprobando es `null`. También es más sencillo: una vez que haya comprobado el tipo, no necesita una comprobación de NULL adicional. Puede comprobar esto en que no existen comprobaciones de NULL en ninguno de los bloques de casos de los ejemplos anteriores: no son necesarias, ya que la coincidencia del patrón de tipo garantiza un valor distinto de NULL.

Declaraciones `var` en expresiones `case`

La introducción de `var` como una de las expresiones de coincidencia presenta nuevas reglas para la coincidencias de patrones.

La primera regla es que la declaración `var` sigue las reglas de inferencia de tipo normal: El tipo se infiere para que sea el tipo estático de la expresión switch. De esa regla, el tipo siempre coincide.

La segunda regla es que una declaración `var` no tiene la comprobación de valores NULL que incluyen otras expresiones de patrón de tipo. Esto significa que la variable puede ser NULL y se necesita una comprobación de valores NULL en ese caso.

Esas dos reglas implican que, en muchos casos, una declaración `var` en una expresión `case` coincide con las mismas condiciones que una expresión `default`. Dado que se prefiere cualquier caso que no sea default al caso `default`, el caso `default` nunca se ejecutará.

> **NOTE**
> El compilador no emite una advertencia en esos casos en que se ha escrito un caso `default` pero nunca se ejecutará. Esto es coherente con el comportamiento actual de la instrucción `switch` donde se han enumerado todos los casos posibles.

La tercera regla presenta usos donde un caso `var` puede resultar útil. Imagine que va a realizar una coincidencia de patrones donde la entrada es una cadena y busca valores de comando conocidos. Podría escribir algo parecido a esto:

```
static object CreateShape(string shapeDescription)
{
    switch (shapeDescription)
    {
        case "circle":
            return new Circle(2);

        case "square":
            return new Square(4);

        case "large-circle":
            return new Circle(12);

        case var o when (o?.Trim().Length ?? 0) == 0:
            // white space
            return null;
        default:
            return "invalid shape description";
    }
}
```

El caso `var` coincide con `null`, la cadena vacía, o cualquier cadena que contenga solo espacios en blanco. Tenga en cuenta que el código anterior usa el operador `?.` para asegurarse de que no genera por accidente una NullReferenceException. El caso `default` controla cualquier otro valor de cadena que no entienda este analizador de comandos.

Este es un ejemplo en que puede que quiera considerar el uso de una expresión de caso `var` que sea distinta de una expresión `default`.

Conclusiones

Las *construcciones de coincidencia de patrones* permiten administrar fácilmente el flujo de control entre distintas variables y tipos que no están relacionados mediante una jerarquía de herencia. También se puede controlar la lógica para usar cualquier condición que se pruebe en la variable. Permite patrones y expresiones que se van a necesitar más a menudo a medida que se crean aplicaciones más distribuidas, donde los datos y los métodos que los manipulan están separados. Observará que las estructuras de forma usadas en este ejemplo no contienen métodos, solo propiedades de solo lectura. La coincidencia de patrones funciona con cualquier tipo de datos. Se escriben expresiones que examinan el objeto y se toman decisiones de flujo de control basadas en esas condiciones.

Compare el código de este ejemplo con el diseño que se obtendría al crear una jerarquía de clases para un elemento `Shape` abstracto y formas derivadas concretas cada una con su propia implementación de un método virtual para calcular el área. A menudo encontrará que las expresiones de coincidencia de patrones pueden ser una herramienta muy útil al trabajar con datos y querer separar las preocupaciones sobre almacenamiento de datos de las preocupaciones sobre comportamiento.

Vea también

- Tutorial: Uso de las características de coincidencia de patrones para ampliar los tipos de datos

Escritura de código C# seguro y eficaz

27/04/2020 • 33 minutes to read • Edit Online

Las nuevas características de C# permiten escribir código seguro verificable con un mejor rendimiento. Si aplica estas técnicas cuidadosamente, habrá menos escenarios que requieran código no seguro. Estas características permiten usar con más facilidad las referencias a tipos de valor como argumentos de método y devoluciones de método. Si aplica estas técnicas de forma segura, minimizará la copia de tipos de valor. Al usar tipos de valor, también podrá minimizar el número de asignaciones y transferencias de recolección de elementos no utilizados.

En gran parte del código de ejemplo de este artículo se utilizan las características agregadas a la versión 7.2 de C#. Para poder usar esas características, tiene que configurar el proyecto para que use la versión 7.2 de C# o una posterior. Para obtener más información sobre cómo establecer la versión del lenguaje, vea cómo configurar la versión del lenguaje.

Este artículo se centra en las técnicas que se deben aplicar para administrar recursos de forma eficaz. Una de las ventajas de utilizar tipos de valor es que a menudo evitan las asignaciones de montón. Pero también hay una desventaja, que es que se copian por valor. Este último aspecto dificulta la optimización de los algoritmos que funcionan en grandes cantidades de datos. Las nuevas características del lenguaje en C# 7.2 proporcionan mecanismos que permiten escribir código seguro y eficaz mediante referencias a tipos de valor. Use estas características acertadamente para minimizar tanto las asignaciones como las operaciones de copia. En este artículo se exploran estas nuevas características.

Este artículo se centra en estas técnicas de administración de recursos:

- Declare un parámetro `readonly struct` para expresar que un tipo es **inmutable**. Esto permite al compilador guardar copias defensivas cuando se usan parámetros `in`.
- Si un tipo no puede ser inmutable, declare miembros `struct` `readonly` para indicar que el miembro no modifica el estado.
- Utilice una devolución `ref readonly` si el valor devuelto es un valor `struct` mayor que IntPtr.Size y la duración del almacenamiento es superior al método que devuelve el valor.
- Si el tamaño de un valor `readonly struct` es mayor que IntPtr.Size, deberá pasarlo como parámetro `in` por motivos de rendimiento.
- No pase nunca un elemento `struct` como parámetro `in` a menos que se declare con el modificador `readonly` o que el método solo llame a los miembros de `readonly` de la estructura. Infringir esta guía puede afectar negativamente al rendimiento y podría provocar un comportamiento oculto.
- Use un valor `ref struct`, o bien un valor `readonly ref struct` como Span<T> o ReadOnlySpan<T>, para que funcione con la memoria como secuencia de bytes.

Estas técnicas obligan a equilibrar dos objetivos contrapuestos en relación con las **referencias** y los **valores**. Las variables que son tipos de referencia contienen una referencia a la ubicación en la memoria. Las variables que son tipos de valor contienen directamente su valor. Estas son diferencias clave importantes para administrar recursos de memoria. Los **tipos de valor** suelen copiarse cuando se pasan a un método o se devuelven desde uno. Este comportamiento incluye la copia del valor de `this` al llamar a los miembros de un tipo de valor. El costo de dicha copia está relacionado con el tamaño del tipo. Los **tipos de referencia** se asignan en el montón administrado. Cada nuevo objeto requiere una nueva asignación, y se debe reclamar posteriormente. Ambas operaciones llevan cierto tiempo. La referencia se copia cuando un tipo de referencia se pasa como argumento a un método o se devuelve desde un método.

En este artículo se utiliza el concepto de ejemplo siguiente de la estructura de punto 3D para explicar estas recomendaciones:

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

Los distintos ejemplos utilizan implementaciones distintas de este concepto.

Declaración de valores struct de solo lectura para tipos de valor inmutables

Al declarar un valor `struct` utilizando el modificador `readonly`, se informa al compilador de que su intención es crear un tipo inmutable. El compilador aplica esa decisión de diseño con las siguientes reglas:

- Todos los miembro de campo deben ser `readonly`.
- Todas las propiedades deben ser de solo lectura, incluidas las implementadas automáticamente.

Estas dos reglas son suficientes para garantizar que ningún miembro de un valor `readonly struct` vaya a modificar el estado de ese valor struct. El valor `struct` es inmutable. La estructura de `Point3D` se podría definir como un valor struct inmutable tal como se muestra en este ejemplo:

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

Siga esta recomendación siempre que su intención de diseño sea crear un tipo de valor inmutable. Cualquier mejora de rendimiento que se aplique es una ventaja adicional. El valor `readonly struct` expresa claramente su intención de diseño.

Declaración de miembros de solo lectura cuando una estructura no pueda ser inmutable

En C# 8.0 y versiones posteriores, cuando un tipo de estructura es mutable, debe declarar los miembros que no provocan la mutación para que sean `readonly`. Plantéese usar una aplicación diferente que necesite una estructura de punto 3D, pero que admita la mutabilidad. La siguiente versión de la estructura de punto 3D agrega el modificador `readonly` solo a los miembros que no modifican la estructura. Siga este ejemplo cuando el diseño deba admitir modificaciones en el tipo de datos struct por parte de algunos miembros, pero desea obtener las ventajas de aplicar readonly en algunos miembros:

```
public struct Point3D
{
    public Point3D(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;
    }

    private double _x;
    public double X
    {
        readonly get => _x;
        set => _x = value;
    }

    private double _y;
    public double Y
    {
        readonly get => _y;
        set => _y = value;
    }

    private double _z;
    public double Z
    {
        readonly get => _z;
        set => _z = value;
    }

    public readonly double Distance => Math.Sqrt(X * X + Y * Y + Z * Z);

    public readonly override string ToString() => $"{X}, {Y}, {Z}";
}
```

En el ejemplo anterior se muestran muchas de las ubicaciones en las que puede aplicar el modificador `readonly`: métodos, propiedades y descriptores de acceso de propiedad. Si usa propiedades implementadas automáticamente, el compilador agrega el modificador `readonly` al descriptor de acceso `get` para las propiedades de lectura y escritura. El compilador agrega el modificador `readonly` a las declaraciones de propiedad implementadas automáticamente para las propiedades con solo un descriptor de acceso `get`.

Agregar el modificador `readonly` a los miembros que no mutan el estado proporciona dos ventajas relacionadas. En primer lugar, el compilador aplica su intención. Ese miembro no puede mutar el estado de la estructura. En segundo lugar, el compilador no crea copias defensivas de parámetros `in` al obtener acceso a un miembro de `readonly`. El compilador puede hacer que esta optimización sea segura, ya que garantiza que un miembro de `readonly` no modifique `struct`.

Uso de instrucciones `ref readonly return` para grandes estructuras en la medida de lo posible

Puede devolver valores por referencia cuando el valor que se devuelva no sea local en el método de devolución. Al devolverlos de esta forma, se copiará solo la referencia, no la estructura. En el ejemplo siguiente, la propiedad `Origin` no puede usar ninguna devolución `ref` porque el valor que se está devolviendo es una variable local:

```
public Point3D Origin => new Point3D(0,0,0);
```

Sin embargo, la siguiente definición de propiedad se puede devolver por referencia porque el valor devuelto es un miembro estático:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}
```

Como no quiere que los autores de llamada modifiquen el origen, debe devolver el valor según su `ref readonly`:

```
public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}
```

Al devolver `ref readonly`, se podrá ahorrar procesos de copia de estructuras mayores y conservar la inmutabilidad de los miembros de datos internos.

En el sitio de llamada, los autores de dicha llamada eligen utilizar la propiedad `Origin` como `ref readonly` o como valor:

```
var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;
```

La primera asignación en el código anterior realiza una copia de la constante `Origin` y asigna esa copia. La segunda asigna una referencia. Tenga en cuenta que el modificador `readonly` debe formar parte de la declaración de la variable. No se puede modificar la referencia a la que alude. Los intentos de hacerlo generarán un error en tiempo de compilación.

El modificador `readonly` es necesario en la declaración de `originReference`.

El compilador exige que el autor de una llamada no pueda modificar la referencia. Los intentos de asignar el valor directamente generan un error en tiempo de compilación. Sin embargo, el compilador no puede saber si algún método de miembro modifica el estado del valor struct. Para asegurarse de que el objeto no se modifique, el compilador crea una copia y llama a las referencias de miembro usando esa copia. Las modificaciones se realizan sobre esa copia defensiva.

Aplicación del modificador `in` en los parámetros `readonly struct` mayores que `System.IntPtr.Size`

La palabra clave `in` complementa las palabras clave `ref` y `out` existentes para pasar argumentos por referencia. La palabra clave `in` especifica que se pasa el argumento por referencia, pero el método llamado no modifica el valor.

Esta novedad proporciona un vocabulario completo para expresar la intención del diseño. Los tipos de valor se copian al pasarlos a un método llamado cuando no se especifica ninguno de los siguientes modificadores en la firma de método. Cada uno de estos modificadores especifica que una variable se pasa por referencia, evitando la copia. Cada modificador expresa un propósito diferente:

- `out`: este método establece el valor del argumento utilizado como este parámetro.
- `ref`: este método puede establecer el valor del argumento utilizado como este parámetro.
- `in`: este método no modifica el valor del argumento utilizado como este parámetro.

Agregue el modificador `in` para pasar un argumento por referencia y declare la intención del diseño de pasar argumentos por referencia para evitar la copia innecesaria. No pretende modificar el objeto usado como ese argumento.

A menudo, esta práctica mejora el rendimiento de los tipos de valor de solo lectura que son mayores que IntPtr.Size. En el caso de los tipos simples (`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool` y `enum`), las posibles mejoras con respecto al rendimiento son mínimas. De hecho, el rendimiento puede empeorar al usar un parámetro de paso por referencia para los tipos menores que IntPtr.Size.

El código siguiente muestra un ejemplo de un método que calcula la distancia entre dos puntos en un espacio 3D.

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Los argumentos son dos estructuras que contienen cada una de ellas tres valores dobles. Un valor doble tiene 8 bytes, por lo que cada argumento es de 24 bytes. Al especificar el modificador `in`, se pasa una referencia de 4 u 8 bytes a esos argumentos, en función de la arquitectura de la máquina. La diferencia de tamaño es pequeña, pero aumenta rápidamente cuando la aplicación llama a este método en un bucle ajustado con muchos valores diferentes.

El modificador `in` complementa también a `out` y `ref` de otras maneras. No puede crear sobrecargas de un método que difiere solo en cuanto a la presencia de `in`, `out` o `ref`. Estas nuevas reglas extienden el mismo comportamiento que siempre se ha definido para los parámetros `out` y `ref`. Como en el caso de los modificadores `out` y `ref`, no se aplica la conversión boxing a los tipos de valor porque se aplica el modificador `in`.

El modificador `in` se puede aplicar a cualquier miembro que toma parámetros: métodos, delegados, expresiones lambda, funciones locales, indexadores u operadores.

Otra de las características de los parámetros `in` es que se pueden usar valores literales o constantes para el argumento en un parámetro `in`. Además, a diferencia de un parámetro `ref` o `out`, no es necesario aplicar el modificador `in` en el sitio de llamada. El código siguiente muestra dos ejemplos de llamada al método `CalculateDistance`. El primero usa dos variables locales pasadas por referencia. El segundo incluye una variable temporal creada como parte de la llamada al método.

```
var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

El compilador tiene varias maneras de aplicar la naturaleza de solo lectura de un argumento `in`. En primer lugar, el método llamado no se puede asignar directamente a un parámetro `in`. No se puede asignar directamente a ningún campo de un parámetro `in` cuando el valor es un tipo `struct`. Además, no se puede pasar un parámetro `in` a ningún método que exija el modificador `ref` o `out`. Estas reglas se aplican a cualquier campo de un parámetro `in`, siempre que el campo sea un tipo `struct` y el parámetro también sea un tipo `struct`. De hecho,

estas reglas se aplican para varios niveles de acceso a miembros, siempre que los tipos en todos los niveles de acceso a miembros sean `structs`. El compilador exige que los tipos `struct` que se pasan como argumentos `in` y sus miembros `struct` sean variables de solo lectura cuando se usan como argumentos para otros métodos.

El uso de parámetros `in` puede evitar los posibles costos de rendimiento que conlleva realizar copias. No cambia la semántica de ninguna llamada al método. Por lo tanto, no es necesario especificar el modificador `in` en el sitio de llamada. Al omitir el modificador `in` en el sitio de llamada, se indica al compilador que está permitido realizar una copia del argumento por estos motivos:

- Hay una conversión implícita, pero no una conversión de identidad desde el tipo de argumento hacia el tipo de parámetro.
- El argumento es una expresión, pero no tiene una variable de almacenamiento conocida.
- Existe una sobrecarga que se diferencia por la presencia o la ausencia de `in`. En ese caso, la sobrecarga por valor es una coincidencia mejor.

Estas reglas son útiles cuando se actualiza código existente para usar argumentos de referencia de solo lectura. En el método llamado, puede llamar a cualquier método de instancia que use parámetros por valor. En esos casos, se crea una copia del parámetro `in`. Dado que el compilador puede crear una variable temporal para cualquier parámetro `in`, también puede especificar valores predeterminados para cualquier parámetro `in`. En este código se especifica el origen (punto 0,0) como valor predeterminado para el segundo punto:

```
private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

Para forzar al compilador que pase argumentos de solo lectura por referencia, especifique el modificador `in` en los argumentos en el sitio de llamada, como se muestra en el este código:

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

Con este comportamiento es más fácil adoptar parámetros `in` con el tiempo en grandes bases de código donde es posible mejorar el rendimiento. Primero, puede agregar el modificador `in` para las firmas de método. Después, puede agregar el modificador `in` en sitios de llamada y crear tipos `readonly struct` para permitir al compilador que evite la creación de copias defensivas de parámetros `in` en más ubicaciones.

La designación del parámetro `in` también se puede usar con tipos de referencia o valores numéricos. Sin embargo, las ventajas de ambos casos son mínimas, si las hay.

Evitar structs mutables como un argumento `in`

En las técnicas descritas anteriormente se explica cómo evitar copias devolviendo referencias y pasando los valores por referencia. Estas técnicas funcionan mejor cuando los tipos de argumento se declaran como tipos `readonly struct`. En caso contrario, el compilador deberá crear **copias defensivas** en muchas situaciones para aplicar la característica de solo lectura en cualquier argumento. Tenga en cuenta el comentario siguiente, en el que se calcula la distancia de un punto 3D respecto del origen:

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

La estructura `Point3D` *no* es un valor struct de solo lectura. Hay seis llamadas de acceso a propiedades en el cuerpo de este método. En el primer examen, es posible que haya pensado que esos accesos son seguros. Al fin y al cabo, un descriptor de acceso `get` no debe modificar el estado del objeto. Sin embargo, no hay ninguna regla de lenguaje que lo exija. Se trata solo de una convención habitual. Cualquier tipo podría implementar un descriptor de acceso `get` que haya modificado el estado interno. Si el compilador no dispone de ninguna garantía relativa al lenguaje, debe crear una copia temporal del argumento antes de llamar a algún miembro no marcado con el modificador `readonly`. El almacenamiento temporal se crea en la pila, los valores del argumento se copian al almacenamiento temporal y el valor se copia en la pila por cada acceso de miembro como argumento `this`. En muchas situaciones, estas copias perjudican tanto el rendimiento que el parámetro de paso por valor es más rápido que el de paso por referencia cuando el tipo de argumento no es un valor `readonly struct`, y el método llama a los miembros no marcados con `readonly`. Si marca todos los métodos que no modifican el estado de struct como `readonly`, el compilador puede determinar con seguridad que no se modifica el estado de struct y no se necesita una copia defensiva.

En lugar de ello, si en el cálculo de distancia se usa la estructura inmutable, `ReadonlyPoint3D`, no se necesitan objetos temporales:

```
private static double CalculateDistance3(in ReadonlyPoint3D point1, in ReadonlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

El compilador genera un código más eficaz cuando se llama a los miembros de un valor `readonly struct`: La referencia `this`, en lugar de una copia del receptor, es siempre un parámetro `in` pasado por referencia al método del miembro. Esta optimización ahorra procesos de copia cuando se utiliza `readonly struct` como argumento `in`.

No se debe pasar un tipo de valor que admite un valor NULL como argumento `in`. El tipo Nullable<T> no se declara como una estructura de solo lectura. Eso significa que el compilador debe generar copias defensivas de cualquier argumento de tipo de valor que acepta valores NULL pasado a un método con el modificador `in` en la declaración de parámetros.

Puede ver un programa de ejemplo en el que se muestran las diferencias de rendimiento con BenchmarkDotNet en el repositorio de ejemplos de GitHub. Compara la transmisión de un valor struct mutable por valor y por referencia con la transmisión de un valor struct inmutable por valor y por referencia. El uso de un valor struct inmutable y de la transmisión por referencia es un proceso más rápido.

Uso de los tipos `ref struct` para trabajar con bloques o memoria en un solo marco de pila

Una característica de lenguaje relacionada es la capacidad de declarar un tipo de valor que debe limitarse a un

solo marco de pila. Esta restricción permite al compilador aplicar una serie de optimizaciones. La principal motivación para esta característica era Span<T> y las estructuras relacionadas. Conseguirá mejoras en el rendimiento a partir de estas optimizaciones usando las API de .NET nuevas y actualizadas que utilizan el tipo Span<T>.

Puede tener requisitos similares al trabajar con memoria creada mediante `stackalloc` o al usar memoria de las API de interoperabilidad. Puede definir sus propios tipos de `ref struct` para esas necesidades.

Tipo de `readonly ref struct`

Si declara una estructura como `readonly ref` se combinan las ventajas y las restricciones de las declaraciones `ref struct` y `readonly struct`. La memoria utilizada por el intervalo de solo lectura está limitada a un solo marco de pila y no se puede modificar.

Conclusiones

El uso de tipos de valor minimiza el número de operaciones de asignación:

- El almacenamiento de los tipos de valor está asignado a la pila en las variables locales y los argumentos de método.
- En cuanto al almacenamiento de tipos de valor que son miembros de otros objetos, está asignado como parte del objeto en cuestión, no como una asignación separada.
- Respecto a los valores devueltos por tipo de valor, están asignados a la pila.

Es necesario contrastar eso con los tipos de referencia en estas mismas situaciones:

- El almacenamiento de los tipos de referencia se asigna al montón en las variables locales y los argumentos de método. La referencia se almacena en la pila.
- El almacenamiento de tipos de referencia que son miembros de otros objetos está asignado por separado en la pila. El objeto contenedor almacena la referencia.
- El almacenamiento de los valores devueltos por tipo de referencia está asignado a la pila. La referencia a ese almacenamiento se guarda en la pila.

La minimización de asignaciones conlleva una serie de renuncias. Se copia más memoria cuando el tamaño del valor `struct` es mayor que el de una referencia. Habitualmente, las referencias son de 64 o 32 bits y dependen de la CPU de la máquina de destino.

Dichas renuncias suelen tener un impacto mínimo en el rendimiento. Sin embargo, en el caso de los valores struct grandes o de las colecciones de mayor tamaño, el impacto sobre el rendimiento es mayor. Dicho impacto puede ser considerable en los bucles estrechos y las rutas de acceso activas de los programas.

Estas mejoras del lenguaje C# están diseñadas para algoritmos en los que el rendimiento es fundamental y la minimización de asignaciones de memoria es un factor principal para lograr el rendimiento necesario. Es posible que no use a menudo estas características en el código que escribe. Sin embargo, estas mejoras se han adoptado a través de .NET. Dado que cada vez son más las API que utilizan estas características, verá que el rendimiento de sus aplicaciones aumenta.

Vea también

- ref (palabra clave)
- Devoluciones y variables locales ref

Árboles de expresión

18/03/2020 • 4 minutes to read • Edit Online

Si ha usado LINQ, tiene experiencia con una extensa biblioteca donde los tipos `Func` forman parte del conjunto de API. (Si no está familiarizado con LINQ, probablemente quiera leer el tutorial de LINQ y el artículo sobre expresiones lambda antes de este). Los *árboles de expresión* proporcionan una interacción más amplia con los argumentos que son funciones.

Escribe argumentos de función, normalmente con expresiones lambda, cuando crea consultas LINQ. En una consulta LINQ habitual, esos argumentos de función se transforman en un delegado que crea el compilador.

Cuando quiera tener una interacción mayor, necesita usar los *árboles de expresión*. Los árboles de expresión representan el código como una estructura que puede examinar, modificar o ejecutar. Estas herramientas le proporcionan la capacidad de manipular el código durante el tiempo de ejecución. Puede escribir código que examine la ejecución de algoritmos o inserte nuevas características. En escenarios más avanzados, puede modificar la ejecución de algoritmos e incluso convertir expresiones de C# en otra forma para su ejecución en otro entorno.

Probablemente ya ha escrito código que use árboles de expresión. Las API de LINQ de Entity Framework admiten árboles de expresión como argumentos para el patrón de expresión de consulta LINQ. Eso permite que Entity Framework convierta la consulta que ha escrito en C# en SQL que se ejecuta en el motor de base de datos. Otro ejemplo es Moq, que es un marco simulado popular de .NET.

En las secciones restantes de este tutorial exploraremos lo que son los árboles de expresión, examinaremos las clases de marco que admiten los árboles de expresión y le mostraremos cómo trabajar con ellos. Obtendrá información sobre cómo leer árboles de expresión, cómo crearlos, cómo crear árboles de expresión modificados y cómo ejecutar el código representado en ellos. Después de esta lectura, estará listo para usar estas estructuras para crear algoritmos muy adaptables.

1. Árboles de expresiones en detalle

   Comprenda la estructura y los conceptos de los *árboles de expresión*.

2. Tipos de marco que admiten árboles de expresión

   Obtenga información sobre las estructuras y las clases que definen y manipulan los árboles de expresión.

3. Ejecución de expresiones

   Obtenga información sobre cómo convertir un árbol de expresión representado como una expresión lambda en un delegado y ejecutar el delegado resultante.

4. Interpretación de expresiones

   Obtenga información sobre cómo recorrer y examinar *árboles de expresión* para entender qué código representa el árbol de expresión.

5. Generación de expresiones

   Obtenga información sobre cómo construir los nodos de un árbol de expresión y crear árboles de expresión.

6. Traducción de expresiones

   Obtenga información sobre cómo crear una copia modificada de un árbol de expresión, o convertir un

árbol de expresión en un formato diferente.

7. Resumen

   Revise la información sobre los árboles de expresión.

Árboles de expresiones en detalle

18/03/2020 • 9 minutes to read • Edit Online

Anterior: Información general

Los árboles de expresiones son estructuras de datos que definen código. Se basan en las mismas estructuras que usa un compilador para analizar el código y generar el resultado compilado. A medida que vaya leyendo este tutorial, observará cierta similitud entre los árboles de expresiones y los tipos usados en las API de Roslyn para compilar analizadores y correcciones de código. (Los analizadores y las correcciones de código son paquetes de NuGet que realizan un análisis estático en código y pueden sugerir posibles correcciones para un desarrollador). Los conceptos son similares y el resultado final es una estructura de datos que permite examinar el código fuente de forma significativa. En cambio, los árboles de expresiones se basan en un conjunto de clases y API totalmente diferente a las API de Roslyn.

Veamos un ejemplo sencillo. Aquí tiene una línea de código:

```
var sum = 1 + 2;
```

Si fuera a analizarlo como un árbol de expresión, el árbol contiene varios nodos. El nodo más externo es una instrucción de declaración de variable con asignación (`var sum = 1 + 2;`). Ese nodo exterior contiene varios nodos secundarios: una declaración de variable, un operador de asignación y una expresión que representa el lado derecho del signo igual. Esa expresión se subdivide aún más en expresiones que representan la operación de suma, y los operandos izquierdo y derecho de la suma.

Vamos a profundizar un poco más en las expresiones que constituyen el lado derecho del signo igual. La expresión es `1 + 2`. Se trata de una expresión binaria. Concretamente, es una expresión binaria de suma. Una expresión binaria de suma tiene dos elementos secundarios, que representan los nodos izquierdo y derecho de la expresión de suma. Aquí, ambos nodos son expresiones constantes: El operador izquierdo es el valor `1` y el operador derecho, el valor `2`.

Visualmente, toda la instrucción es un árbol: puede empezar en el nodo raíz y desplazarse a cada uno de los nodos del árbol para ver el código que compone la instrucción:

- Instrucción de declaración de variable con asignación (`var sum = 1 + 2;`)
  - Declaración de tipo de variable implícita (`var sum`)
    - Palabra clave var implícita (`var`)
    - Declaración de nombre de variable (`sum`)
  - Operador de asignación (`=`)
  - Expresión binaria de suma (`1 + 2`)
    - Operando izquierdo (`1`)
    - Operador de suma (`+`)
    - Operando derecho (`2`)

Esto puede parecer complicado, pero resulta muy eficaz. Siguiendo el mismo proceso, puede descomponer expresiones mucho más complicadas. Tomemos esta expresión como ejemplo:

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

La expresión anterior también es una declaración de variable con una asignación. En este caso, el lado derecho de la asignación es un árbol mucho más complicado. No voy a descomponer esta expresión, pero tenga en cuenta lo que podrían ser los distintos nodos. Hay llamadas de método que usan el objeto actual como un receptor, una que tiene un receptor `this` explícito y otra que no. Hay llamadas de método que usan otros objetos de receptor, así como argumentos constantes de tipos diferentes. Y, por último, hay un operador binario de suma. Según el tipo de valor devuelto de `SecretSauceFunction()` o `MoreSecretSauce()`, ese operador binario de suma puede ser una llamada de método a un operador de suma invalidado, que se resuelva en una llamada de método estático al operador binario de suma definido para una clase.

A pesar de esta aparente complejidad, la expresión anterior crea una estructura de árbol por la que se puede navegar con tanta facilidad como en el primer ejemplo. Puede seguir recorriendo los nodos secundarios para buscar nodos hoja en la expresión. Los nodos primarios tendrán referencias a sus elementos secundarios, y cada nodo tiene una propiedad que describe de qué tipo es.

La estructura de los árboles de expresiones es muy coherente. Una vez que conozca los aspectos básicos, podrá entender incluso el código más complejo cuando esté representado como un árbol de expresión. La elegancia de la estructura de datos explica cómo el compilador de C# puede analizar los programas de C# más complejos y crear resultados correctos a partir de código fuente complicado.

Una vez que esté familiarizado con la estructura de los árboles de expresiones, verá que los conocimientos que ha adquirido le permiten trabajar rápidamente con muchos escenarios más avanzados. Los árboles de expresiones ofrecen posibilidades increíbles.

Además de traducir algoritmos para ejecutarlos en otros entornos, se pueden usar árboles de expresiones para que resulte más fácil escribir algoritmos que inspeccionen el código antes de ejecutarlo. Puede escribir un método cuyos argumentos sean expresiones y, luego, examinar esas expresiones antes de ejecutar el código. El árbol de expresión es una representación completa del código: puede ver los valores de cualquier subexpresión. Puede ver los nombres de propiedad y método. Puede ver el valor de las expresiones constantes. También puede convertir un árbol de expresión en un delegado ejecutable y ejecutar el código.

Las API de los árboles de expresiones permiten crear árboles que representan casi cualquier construcción de código válida. En cambio, para que todo resulte lo más sencillo posible, algunas expresiones de C# no se pueden crear en un árbol de expresión. Un ejemplo son las expresiones asincrónicas (mediante las palabras clave `async` y `await`). Si necesita algoritmos asincrónicos, tendría que manipular los objetos `Task` directamente, en lugar de confiar en la compatibilidad del compilador. Otro ejemplo es en la creación de bucles. Normalmente, puede crearlos usando bucles `for`, `foreach`, `while` o `do`. Como verá más adelante en esta serie, las API de los árboles de expresiones admiten una expresión de bucle individual, con expresiones `break` y `continue` que controlan la repetición del bucle.

Lo único lo que no se puede hacer es modificar un árbol de expresión. Los árboles de expresiones son estructuras de datos inmutables. Si quiere mutar (cambiar) un árbol de expresión, debe crear un nuevo árbol que sea una copia del original, pero con los cambios que quiera.

Siguiente: Tipos de marco que admiten árboles de expresión

Tipos de marco que admiten árboles de expresión

18/03/2020 • 6 minutes to read • Edit Online

Anterior: Árboles de expresiones en detalle

Hay una amplia lista de clases en .NET Core Framework que funcionan con árboles de expresiones. En System.Linq.Expressions puede ver la lista completa. En lugar de analizarla al completo, vamos a explicar cómo se han diseñado las clases del marco.

En el diseño del lenguaje, una expresión es un cuerpo de código que se evalúa y devuelve un valor. Las expresiones pueden ser muy sencillas: la expresión constante `1` devuelve el valor constante de 1. También pueden ser más complicadas: la expresión `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` devuelve una raíz de una ecuación cuadrática (en el caso en el que la ecuación tenga una solución).

Todo empieza con System.Linq.Expression

Una de las complejidades de trabajar con árboles de expresiones es que muchos tipos de expresiones distintos son válidos en muchos lugares de los programas. Piense en una expresión de asignación. El lado derecho de una asignación podría ser un valor constante, una variable, una expresión de llamada de método u otros elementos. Esa flexibilidad del lenguaje significa que puede encontrarse con muchos tipos de expresiones diferentes en cualquier parte de los nodos de un árbol al atravesar un árbol de expresión. Por lo tanto, lo más sencillo consiste en trabajar con el tipo de expresión base, siempre que sea posible. En cambio, en ocasiones necesitará saber más. La clase de expresión base contiene una propiedad `NodeType` para ello. Esta devuelve un elemento `ExpressionType` , que es una enumeración de tipos de expresiones posibles. Una vez que sepa el tipo del nodo, puede convertirlo en ese tipo y realizar acciones específicas sabiendo el tipo del nodo de expresión. Puede buscar determinados tipos de nodo y, luego, trabajar con las propiedades específicas de ese tipo de expresión.

Por ejemplo, este código imprimirá el nombre de una variable para una expresión de acceso a la variable. He seguido el procedimiento que consiste en comprobar el tipo de nodo, convertirlo en una expresión de acceso a la variable y después comprobar las propiedades del tipo de expresión específico:

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.First();

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Crear árboles de expresiones

La clase `System.Linq.Expression` también contiene muchos métodos estáticos para crear expresiones. Estos métodos crean un nodo de expresión al usar los argumentos proporcionados para sus elementos secundarios. De esta manera, se crea una expresión a partir de sus nodos hoja. Por ejemplo, este código genera una expresión de agregar:

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

En este sencillo ejemplo puede ver que hay muchos tipos implicados a la hora de crear árboles de expresiones y trabajar con ellos. Esta complejidad resulta necesaria para proporcionar las capacidades del vocabulario variado que ofrece el lenguaje C#.

Navegar por las API

Hay tipos de nodos de expresión que se asignan a casi todos los elementos de sintaxis del lenguaje C#. Cada tipo tiene métodos específicos para ese tipo de elemento del lenguaje. Es mucha información como para recordarla toda. En lugar de intentar memorizar todo, estas son las técnicas que uso para trabajar con árboles de expresiones:

1. Fíjese en los miembros de la enumeración `ExpressionType` para determinar los posibles nodos que debe examinar. Esto resulta muy útil cuando quiere atravesar y comprender un árbol de expresión.
2. Fíjese en los miembros estáticos de la clase `Expression` para crear una expresión. Esos métodos pueden crear cualquier tipo de expresión a partir de un conjunto de sus nodos secundarios.
3. Fíjese en la clase `ExpressionVisitor` para crear un árbol de expresión modificado.

Encontrará más información a medida que observe cada una de esas tres áreas. Siempre encontrará lo que necesita empezando con uno de esos tres pasos.

Siguiente: Ejecutar árboles de expresión

Ejecución de árboles de expresión

18/03/2020 • 11 minutes to read • Edit Online

Anterior: Tipos de marco que admiten árboles de expresión

Un *árbol de expresión* es una estructura de datos que representa un código. No es código compilado y ejecutable. Si quiere ejecutar el código de .NET que se representa mediante un árbol de expresión, debe convertirlo en instrucciones de lenguaje intermedio ejecutables.

Expresiones lambda a funciones

Puede convertir cualquier objeto LambdaExpression o cualquier tipo derivado de LambdaExpression en lenguaje intermedio ejecutable. Otros tipos de expresión no se pueden convertir directamente a código. Esta restricción tiene poco efecto en la práctica. Las expresiones lambda son los únicos tipos de expresiones que podría querer ejecutar mediante la conversión a lenguaje intermedio (IL) ejecutable. (Piense lo que significaría ejecutar directamente una `ConstantExpression`. ¿Tendría algún significado útil?). Cualquier árbol de expresión que es una `LambdaExpression` o un tipo derivado de `LambdaExpression` se puede convertir a lenguaje intermedio. El tipo de expresión `Expression<TDelegate>` es el único ejemplo concreto en las bibliotecas de .NET Core. Se usa para representar una expresión que se asigna a cualquier tipo de delegado. Dado que este tipo se asigna a un tipo de delegado, .NET puede examinar la expresión y generar el lenguaje intermedio de un delegado adecuado que coincida con la firma de la expresión lambda.

En la mayoría de los casos, esto crea una asignación simple entre una expresión y su delegado correspondiente. Por ejemplo, un árbol de expresión que se representa por `Expression<Func<int>>` se convertiría a un delegado del tipo `Func<int>`. Para una expresión lambda con cualquier tipo de valor devuelto y lista de argumentos, existe un tipo de delegado que es el tipo de destino para el código ejecutable representado por esa expresión lambda.

El tipo `LambdaExpression` contiene los miembros `Compile` y `CompileToMethod` que se usarían para convertir un árbol de expresión en código ejecutable. El método `Compile` crea un delegado. El método `CompileToMethod` actualiza un objeto `MethodBuilder` con el lenguaje intermedio que representa la salida compilada del árbol de expresión. Tenga en cuenta que `CompileToMethod` solo está disponible en el marco de trabajo de escritorio completo, no en .NET Core.

Como opción, también puede proporcionar un `DebugInfoGenerator` para que reciba la información de depuración de símbolos para el objeto de delegado generado. Esto le permite convertir el árbol de expresión en un objeto de delegado y disponer de información de depuración completa sobre el delegado generado.

Para convertir una expresión en un delegado se usaría el siguiente código:

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

Observe que el tipo de delegado se basa en el tipo de expresión. Debe conocer el tipo de valor devuelto y la lista de argumentos si quiere usar el objeto de delegado de una forma fuertemente tipada. El método `LambdaExpression.Compile()` devuelve el tipo `Delegate`. Tendrá que convertirlo al tipo de delegado correcto para que las herramientas de tiempo de compilación comprueben la lista de argumentos del tipo de valor devuelto.

Ejecución y duraciones

El código se ejecuta mediante la invocación del delegado que se crea al llamar a `LambdaExpression.Compile()`. Puede verlo encima de donde `add.Compile()` devuelve un delegado. Invocar ese delegado, mediante una llamada a `func()` ejecuta el código.

Ese delegado representa el código en el árbol de expresión. Se puede conservar el identificador a ese delegado e invocarlo más adelante. No es necesario compilar el árbol de expresión cada vez que se quiera ejecutar el código que representa. (Recuerde que los árboles de expresión son inmutables y que compilar el mismo árbol de expresión más adelante creará un delegado que ejecuta el mismo código).

No es aconsejable intentar crear cualquier mecanismo de almacenamiento en caché más sofisticado para aumentar el rendimiento evitando llamadas innecesarias de compilación. La comparación de dos árboles de expresión arbitrarios para determinar si representan el mismo algoritmo también consumirá mucho tiempo de ejecución. Probablemente comprobará que el tiempo de proceso que se ahorra al evitar llamadas adicionales a `LambdaExpression.Compile()` será consumido por el tiempo de ejecución de código que determina que dos árboles de expresión diferentes devuelven el mismo código ejecutable.

Advertencias

Compilar una expresión lambda en un delegado e invocar ese delegado es una de las operaciones más simples que se pueden realizar con un árbol de expresión. Pero incluso con esta sencilla operación, hay advertencias que debe conocer.

Las expresiones lambda crean clausuras sobre las variables locales a las que se hace referencia en la expresión. Debe garantizar que las variables que formarían parte del delegado se pueden usar en la ubicación desde la que se llama a `Compile`, y cuando se ejecuta el delegado resultante.

En general, el compilador se asegurará de que esto es cierto. Pero si la expresión tiene acceso a una variable que implementa `IDisposable`, es posible que el código deseche el objeto mientras se sigue manteniendo en el árbol de expresión.

Por ejemplo, este código funciona bien porque `int` no implementa `IDisposable`:

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

El delegado capturó una referencia a la variable local `constant`. Esa variable es accesible en cualquier momento posterior, cuando se ejecuta la función devuelta por `CreateBoundFunc`.

Pero considere esta clase (bastante artificiosa) que implementa `IDisposable`:

```
public class Resource : IDisposable
{
    private bool isDisposed = false;
    public int Argument
    {
        get
        {
            if (!isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        isDisposed = true;
    }
}
```

Si se usa en una expresión como se muestra a continuación, obtendrá una `ObjectDisposedException` al ejecutar el código al que hace referencia la propiedad `Resource.Argument`:

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument + b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

El delegado devuelto por este método se clausuró sobre el objeto `constant`, que se eliminó. (Se eliminó porque se declaró en una instrucción `using`).

Ahora, al ejecutar el delegado devuelto por este método, se producirá una excepción `ObjectDisposedException` en el punto de ejecución.

Parece extraño tener un error en tiempo de ejecución que representa una construcción de tiempo de compilación, pero es el mundo al que se entra cuando se trabaja con árboles de expresión.

Hay una gran cantidad de permutaciones de este problema, por lo que resulta difícil ofrecer instrucciones generales para evitarlo. Tenga cuidado al obtener acceso a las variables locales al definir expresiones y al obtener acceso al estado en el objeto actual (representado por `this`) al crear un árbol de expresión que pueda ser devuelto por una API pública.

El código de la expresión puede hacer referencia a métodos o propiedades de otros ensamblados. Ese ensamblado debe ser accesible cuando se define la expresión, cuando se compila y cuando se invoca el delegado resultante. En los casos en los que no esté presente, se producirá una excepción `ReferencedAssemblyNotFoundException`.

Resumen

Los árboles de expresión que representan expresiones lambda se pueden compilar para crear un delegado que se puede ejecutar. Esto proporciona un mecanismo para ejecutar el código representado por un árbol de expresión.

El árbol de expresión representa el código que se ejecutaría para cualquier construcción que se cree. Mientras que el entorno donde se compile y ejecute el código coincida con el entorno donde se crea la expresión, todo funciona según lo esperado. Cuando eso no sucede, los errores son muy predecibles y se detectarán en las primeras pruebas de cualquier código que use los árboles de expresión.

Siguiente: Interpretación de expresiones

Interpretación de expresiones

29/06/2020 • 20 minutes to read • Edit Online

Anterior: Ejecución de expresiones

Ahora, vamos a escribir código para examinar la estructura de un *árbol de expresión*. Cada nodo de un árbol de expresión será un objeto de una clase derivada de `Expression`.

Ese diseño hace que la visita de todos los nodos de un árbol de expresión sea una operación recursiva relativamente sencilla. La estrategia general es comenzar en el nodo raíz y determinar qué tipo de nodo es.

Si el tipo de nodo tiene elementos secundarios, visítelos recursivamente. En cada nodo secundario, repita el proceso que ha usado en el nodo raíz: determine el tipo, y si el tipo tiene elementos secundarios, visite cada uno de ellos.

Examinar una expresión sin elementos secundarios

Empecemos visitando cada nodo en un árbol de expresión sencillo. Aquí se muestra el código que crea una expresión constante y, después, examina sus propiedades:

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

Esto imprimirá lo siguiente:

```
This is an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Ahora, vamos a escribir el código que examinará esta expresión y también algunas propiedades importantes sobre este. Aquí se muestra el código:

Examinar una expresión de adición sencilla

Comencemos con el ejemplo de adición de la instrucción de esta sección.

```
Expression<Func<int>> sum = () => 1 + 2;
```

> No estoy usando `var` para declarar este árbol de expresión, y no es posible porque el lado derecho de la asignación tiene un tipo implícito.

El nodo raíz es `LambdaExpression`. Para obtener el código que nos interesa en el lado derecho del operador `=>`, hay que buscar uno de los elementos secundarios de `LambdaExpression`. Haremos esto con todas las expresiones de esta sección. El nodo primario nos ayuda a encontrar el tipo de valor devuelto de `LambdaExpression`.

Para examinar cada nodo de esta expresión, necesitaremos visitar recursivamente varios nodos. Aquí se muestra una primera implementación sencilla:

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count} arguments. They are:");
foreach(var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type: {argumentExpression.Type.ToString()}, Name:
{argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType} expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right= (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name: {right.Name}");
```

Este ejemplo imprime el siguiente resultado:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
        Parameter Type: System.Int32, Name: a
        Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
        Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
        Parameter Type: System.Int32, Name: b
```

Observará muchas repeticiones en el ejemplo de código anterior. Vamos a limpiarlo y a crear un visitante del nodo de expresión con una finalidad más general. Para ello vamos a necesitar escribir un algoritmo recursivo. Cualquier nodo puede ser de un tipo que pueda tener elementos secundarios. Cualquier nodo que tenga elementos secundarios necesita que los visitemos y determinemos cuál es ese nodo. Aquí se muestra una versión limpia que usa la recursividad para visitar las operaciones de adición:

```
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node)
    {
        this.node = node;
    }

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:
```

```
                return new ParameterVisitor((ParameterExpression)node);
            case ExpressionType.Add:
                return new BinaryVisitor((BinaryExpression)node);
            default:
                Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
                return default(Visitor);
        }
    }
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name == null) ? "<null>" :
node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }
```

```
    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type.ToString()}, Name: {node.Name}, ByRef: {node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is {node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is {node.Value}");
    }
}
```

Este algoritmo es la base de un algoritmo que puede visitar cualquier `LambdaExpression` arbitrario. Hay muchas vulnerabilidades, concretamente que el código que he creado solo busca una muestra muy pequeña de los posibles conjuntos de nodos de árbol de expresión que puede encontrar. En cambio, todavía puede aprender bastante de lo que genera. (El caso predeterminado en el método `Visitor.CreateFromExpression` imprime un mensaje a la consola de error cuando se detecta un nuevo tipo de nodo. De esta manera, sabe que se va a agregar un nuevo tipo de expresión).

Cuando ejecuta este visitante en la expresión de adición que se muestra arriba, obtiene el siguiente resultado:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: b, ByRef: False
```

Ahora que ha creado una implementación de visitante más general, puede visitar y procesar muchos más tipos de expresiones diferentes.

Examinar una expresión de adición con muchos niveles

Vamos a probar un ejemplo más complicado, todavía limitando los tipos de nodo solo a los de adición:

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Antes de que ejecute esto en el algoritmo de visitante, haga un ejercicio de reflexión para calcular cuál podría ser el

resultado. Recuerde que el operador `+` es un *operador binario*: debe tener dos elementos secundarios, que representen los operandos izquierdo y derecho. Existen varias maneras posibles de construir un árbol que pueda ser correcto:

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

Puede ver la separación en dos respuestas posibles para resaltar la más prometedora. La primera representa las expresiones *asociativas por la derecha*. La segunda representa las expresiones *asociativas por la izquierda*. La ventaja de los dos formatos es que el formato escala a cualquier número arbitrario de expresiones de adición.

Si ejecuta esta expresión a través del visitante, verá este resultado y comprobará que la expresión de adición simple es *asociativa por la izquierda*.

Para ejecutar este ejemplo, y ver el árbol de expresión completo, tuve que realizar un cambio en el árbol de expresión de origen. Cuando el árbol de expresión contiene todas las constantes, el árbol resultante simplemente contiene el valor constante de `10`. El compilador realiza toda la adición y reduce la expresión a su forma más simple. Simplemente con agregar una variable a la expresión es suficiente para ver el árbol original:

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Cree un visitante para esta suma y ejecute el visitante para ver este resultado:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This binary expression is a Add expression
                        The Left argument is:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                        The Right argument is:
                                This is an Parameter expression type
                                Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
        The Right argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 4
```

También puede ejecutar cualquiera de los otros ejemplos a través del código de visitante y ver qué árbol representa. Aquí se muestra un ejemplo de la expresión `sum3` anterior (con un parámetro adicional para evitar que el compilador calcule la constante):

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Aquí se muestra el resultado del visitante:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 1
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: b, ByRef: False
```

Tenga en cuenta que los paréntesis no forman parte del resultado. No existen nodos en el árbol de expresión que representen los paréntesis en la expresión de entrada. La estructura del árbol de expresión contiene toda la información necesaria para comunicar la precedencia.

Ampliar a partir de este ejemplo

El ejemplo trata solo los árboles de expresión más básicos. El código que ha visto en esta sección solo controla enteros constantes y el operador binario `+`. Como último ejemplo, vamos a actualizar el visitante para que controle una expresión más complicada. Hagamos que funcione para esto:

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

Este código representa una posible implementación para la función *factorial* matemática. La manera en que he escrito este código resalta dos limitaciones en la creación de árboles de expresión asignando expresiones lambda a las expresiones. En primer lugar, las expresiones lambda de instrucción no están permitidas. Eso significa que no puedo usar bucles, bloques, instrucciones IF/ELSE ni otras estructuras de control comunes en C#. Estoy limitado al uso de expresiones. En segundo lugar, no puedo llamar recursivamente a la misma expresión. Podría si ya fuera un delegado, pero no puedo llamarla en su forma de árbol de expresión. En la sección sobre la creación de árboles de expresión, obtendrá las técnicas para superar estas limitaciones.

En esta expresión, encontrará nodos de todos estos tipos:

1. Equal (expresión binaria)

2. Multiply (expresión binaria)
3. Conditional (la expresión ? :)
4. Expresión de llamada a método (con la llamada a `Range()` y `Aggregate()`)

Una manera de modificar el algoritmo de visitante es seguir ejecutándolo y escribir el tipo de nodo cada vez que llegue a su cláusula `default`. Después de varias iteraciones, habrá visto cada uno de los nodos potenciales. Después, tiene todo lo que necesita. El resultado será similar al siguiente:

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

ConditionalVisitor y MethodCallVisitor procesan esos dos nodos:

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType} expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is {methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach(var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

Y el resultado del árbol de expresión será:

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
The expression body is:
        This expression is a Conditional expression
        The Test for this expression is:
                This binary expression is a Equal expression
                The Left argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: n, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 0
        The True clause for this expression is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
        The False clause for this expression is:
                This expression is a Call expression
                This is a static method call
                The method name is System.Linq.Enumerable.Aggregate
                The Arguments are:
                        This expression is a Call expression
                        This is a static method call
                        The method name is System.Linq.Enumerable.Range
                        The Arguments are:
                                This is an Constant expression type
                                The type of the constant value is System.Int32
                                The value of the constant value is 1
                                This is an Parameter expression type
                                Type: System.Int32, Name: n, ByRef: False
                        This expression is a Lambda expression type
                        The name of the lambda is <null>
                        The return type is System.Int32
                        The expression has 2 arguments. They are:
                                This is an Parameter expression type
                                Type: System.Int32, Name: product, ByRef: False
                                This is an Parameter expression type
                                Type: System.Int32, Name: factor, ByRef: False
                        The expression body is:
                                This binary expression is a Multiply expression
                                The Left argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: product, ByRef: False
                                The Right argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: factor, ByRef: False
```

Ampliar la biblioteca de ejemplo

Los ejemplos de esta sección muestran las técnicas principales para visitar y examinar nodos de un árbol de expresión. He pasado por alto muchas acciones que puede necesitar para concentrarme en las tareas principales a la hora de visitar y acceder a los nodos de un árbol de expresión.

En primer lugar, los visitantes solo controlan constantes que son enteros. Los valores constantes pueden ser cualquier otro tipo numérico, y el lenguaje de C# admite conversiones y promociones entre esos tipos. Una versión más sólida de este código reflejará todas esas capacidades.

Incluso en el último ejemplo se reconoce un subconjunto de los tipos de nodo posibles. Todavía puede proporcionarle muchas expresiones que provocarán que se produzca un error. En .NET Standard se incluye una

implementación completa con el nombre ExpressionVisitor que puede controlar todos los tipos de nodo posibles.

Por último, la biblioteca que he usado en este artículo se ha creado con fines de demostración y aprendizaje. No está optimizada. La he escrito para dejar claro las estructuras que he usado y para resaltar las técnicas usadas para visitar los nodos y analizar lo que hay ahí. Una implementación de producción prestaría más atención al rendimiento.

Incluso con esas limitaciones, se encuentra en el camino correcto para escribir algoritmos que lean y comprendan árboles de expresión.

Siguiente: Crear árboles de expresión

Crear árboles de expresión

18/03/2020 • 9 minutes to read • Edit Online

Anterior: Interpretación de expresiones

Hasta ahora, todos los árboles de expresión que ha visto se han creado con el compilador de C#. Todo lo que tenía que hacer era crear una expresión lambda que se asignaba a una variable de tipo `Expression<Func<T>>` o de algún tipo similar. Esa no es la única manera de crear un árbol de expresión. En muchos escenarios, puede que necesite crear una expresión en memoria en tiempo de ejecución.

Crear árboles de expresión es complicado por el hecho de que esos árboles de expresión son inmutables. Inmutable significa que debe crear el árbol desde las hojas hasta la raíz. Las API que usará para crear los árboles de expresión reflejan este hecho: los métodos que usará para crear un nodo toman todos sus elementos secundarios como argumentos. Veamos algunos ejemplos para mostrarle las técnicas.

Crear nodos

Comencemos por algo relativamente sencillo de nuevo. Usaremos la expresión de adición con la que he estado trabajando en estas secciones:

```
Expression<Func<int>> sum = () => 1 + 2;
```

Para crear ese árbol de expresión, debe crear los nodos de hoja. Los nodos de hoja son constantes, por lo que puede usar el método `Expression.Constant` para crear los nodos:

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Después, creará la expresión de adición:

```
var addition = Expression.Add(one, two);
```

Una vez que tenga la expresión de adición, puede crear la expresión lambda:

```
var lambda = Expression.Lambda(addition);
```

Esta es una expresión lambda muy sencilla porque no contiene argumentos. Posteriormente en esta sección, verá cómo asignar argumentos a parámetros y crear expresiones más complicadas.

Para las expresiones que son tan simples como esta, puede combinar todas las llamadas en una sola instrucción:

```
var lambda = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Crear un árbol

Estos son los aspectos básicos para crear un árbol de expresión en la memoria. Los árboles más complejos implican normalmente más tipos de nodo y más nodos en el árbol. Vamos a analizar un ejemplo más y a mostrar dos tipos de nodo que creará normalmente al crear árboles de expresión: los nodos de argumentos y los nodos de llamada al método.

Vamos a crear un árbol de expresión para crear esta expresión:

```
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

Comenzará creando expresiones de parámetro para `x` y `y`:

```
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

La creación de expresiones de adición y multiplicación sigue el patrón que ya ha visto:

```
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

Después, necesita crear una expresión de llamada al método para la llamada a `Math.Sqrt`.

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });
var distance = Expression.Call(sqrtMethod, sum);
```

Y, por último, coloque la llamada al método en una expresión lambda y asegúrese de definir los argumentos en dicha expresión:

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

En este ejemplo más complejo, verá un par de técnicas más que necesitará a menudo para crear árboles de expresión.

Primero, necesita crear los objetos que representan parámetros o variables locales antes de usarlos. Una vez que haya creado esos objetos, puede usarlos en su árbol de expresión siempre que los necesite.

Después, necesita usar un subconjunto de las API de reflexión para crear un objeto `MethodInfo`, de manera que pueda crear un árbol de expresión para tener acceso a ese método. Debe limitarse al subconjunto de las API de reflexión que están disponibles en la plataforma de .NET Core. De nuevo, estas técnicas se extenderán a otros árboles de expresión.

Crear código en profundidad

No está limitado en lo que puede crear con estas API. En cambio, cuánto más complicado sea el árbol de expresión que quiera crear, más difícil será la administración y la lectura del código.

Vamos a crear un árbol de expresión que sea equivalente a este código:

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

Observe anteriormente que no he creado el árbol de expresión, sino simplemente el delegado. Con la clase `Expression` no puede crear expresiones lambda de instrucción. Aquí se muestra el código necesario para crear la misma función. Es complicado por el hecho de que no existe una API para crear un bucle `while`, en su lugar necesita crear un bucle que contenga una prueba condicional y un destino de la etiqueta para salir del bucle.

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

El código para crear el árbol de expresión para la función factorial es bastante más largo, más complicado y está lleno de etiquetas, instrucciones Break y otros elementos que nos gustaría evitar en nuestras tareas de codificación diarias.

En esta sección, también he actualizado el código del visitante para visitar cada nodo de este árbol de expresión y escribir información sobre los nodos que se crean en este ejemplo. Puede ver o descargar el código de ejemplo en el repositorio dotnet/docs de GitHub. Pruébelo compilando y ejecutando los ejemplos. Para obtener instrucciones de descarga, vea Ejemplos y tutoriales.

Examinar las API

Las API del árbol de expresión son algunas de las más difíciles para navegar en .NET Core, pero no pasa nada. Su finalidad es una tarea más compleja: escribir código que genere código en tiempo de ejecución. Son inevitablemente complicadas a la hora de proporcionar un equilibrio entre la compatibilidad con todas las

estructuras de control disponibles en el lenguaje de C# y mantener el área expuesta de las API tan pequeña como sea razonable. Este equilibrio significa que muchas estructuras de control se representan no por sus construcciones de C#, sino por construcciones que representan la lógica subyacente que genera el compilador desde estas construcciones de nivel superior.

Además, en este momento, existen expresiones de C# que no pueden crearse directamente con métodos de clase `Expression`. En general, estos serán las expresiones y los operadores más recientes que se han agregado a C# 5 y C# 6. (Por ejemplo, las expresiones `async` no pueden crearse y el operador `?.` nuevo no puede crearse directamente).

Siguiente: Traducción de árboles de expresión

Traslado de árboles de expresión

18/03/2020 • 10 minutes to read • Edit Online

Anterior: Generación de expresiones

En esta última sección, obtendrá información sobre cómo visitar cada nodo en un árbol de expresión, mientras se crea una copia modificada de ese árbol de expresión. Se trata de las técnicas que se van a usar en dos escenarios importantes. La primera consiste en comprender los algoritmos que se expresan mediante un árbol de expresión para poder trasladarlo a otro entorno. La segunda es cuando se quiere cambiar el algoritmo que se creó. Esto podría ser para agregar el registro, interceptar las llamadas de método y realizar un seguimiento de ellas, o con otros fines.

Trasladar es visitar

El código que se compila para trasladar un árbol de expresión es una extensión de lo que ya se vio para visitar todos los nodos de un árbol. Al trasladar un árbol de expresión, se visitan todos los nodos y mientras se visitan, se crea el árbol nuevo. El nuevo árbol puede contener referencias a los nodos originales o a nodos nuevos que se colocaron en el árbol.

Vamos a verlo en acción mediante la visita a un árbol de expresión y la creación de un árbol nuevo con varios nodos de reemplazo. En este ejemplo, se van a sustituir todas las constantes con una constante que es diez veces mayor. De lo contrario, el árbol de expresión se mantendrá intacto. En lugar de leer el valor de la constante y reemplazarlo con una constante nueva, este cambio se hará reemplazando el nodo constante con un nuevo nodo que realiza la multiplicación.

Aquí, una vez que se encuentre un nodo constante, se crea un nuevo nodo de multiplicación cuyos elementos secundarios son la constante original y la constante `10`:

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

Al reemplazar el nodo original con el sustituto, se crea un árbol nuevo que contiene las modificaciones. Se puede comprobar mediante la compilación y ejecución del árbol reemplazado.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

La creación de un árbol nuevo es una combinación de visitar los nodos del árbol existente y crear nodos nuevos e insertarlos en el árbol.

En este ejemplo se muestra la importancia de la inmutabilidad de los árboles de expresión. Observe que el nuevo árbol creado anteriormente contiene una mezcla de los nodos recién creados y los nodos del árbol existente. Es seguro, ya que no se pueden modificar los nodos en el árbol existente. Esto puede producir eficiencias de memoria significativas. Los mismos nodos se pueden usar en un árbol o en varios árboles de expresión. Dado que los nodos no se pueden modificar, se puede volver a usar el mismo nodo siempre que sea necesario.

Recorrer y ejecutar una adición

Vamos a comprobarlo mediante la creación de un segundo visitante que recorre el árbol de nodos de adición y calcula el resultado. Para ello, se pueden realizar algunas modificaciones en el visitante visto hasta el momento. En esta nueva versión, el visitante devolverá la suma parcial de la operación de adición hasta este punto. Para una expresión constante, es simplemente el valor de la expresión constante. Para una expresión de adición, el resultado es la suma de los operandos izquierdo y derecho, una vez que se recorren esos árboles.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three= Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so we can call it
// from itself recursively:
Func<Expression, int> aggregate = null;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
    aggregate(((BinaryExpression)exp).Left) + aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

Aquí hay gran cantidad de código, pero los conceptos son muy accesibles. Este código visita los elementos secundarios en una primera búsqueda de profundidad. Cuando encuentra un nodo constante, el visitante devuelve el valor de la constante. Tras la visita a los dos elementos secundarios por parte del visitante, ambos elementos habrán obtenido la suma calculada para ese subárbol. El nodo de adición ahora puede calcular la suma. Una vez que se visiten todos los nodos en el árbol de expresión, se habrá calculado la suma. Se puede hacer el seguimiento de la ejecución ejecutando el ejemplo en el depurador y realizando el seguimiento de la ejecución.

Vamos a facilitar el seguimiento de cómo se analizan los nodos y cómo se calcula la suma mediante el recorrido del árbol. Esta es una versión actualizada del método agregado que incluye gran cantidad de información de seguimiento:

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

Al ejecutarla en la misma expresión, produce el siguiente resultado:

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Realice el seguimiento del resultado y siga el código anterior. Debería poder averiguar cómo el código visita cada nodo y calcula la suma mientras recorre el árbol y busca la suma.

Ahora, veremos una ejecución diferente, con la expresión proporcionada por `sum1`:

```
Expression<Func<int> sum1 = () => 1 + (2 + (3 + 4));
```

Este es el resultado de examinar esta expresión:

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

Aunque la respuesta final es la misma, el recorrido del árbol es completamente diferente. Los nodos se recorren en un orden diferente, porque el árbol se construyó con diferentes operaciones que se producen en primer lugar.

Obtener más información

En este ejemplo se muestra un pequeño subconjunto del código que se compilaría para recorrer e interpretar los algoritmos representados por un árbol de expresión. Para obtener una descripción completa de todo el trabajo necesario para compilar una biblioteca de propósito general que traduce árboles de expresión a otro lenguaje, lea esta serie de Matt Warren. Describe en detalle cómo traducir cualquier código que es posible encontrar en un árbol de expresión.

Espero que haya visto la verdadera eficacia de los árboles de expresión. Puede examinar un conjunto de código, realizar los cambios que quiera en ese código y ejecutar la versión modificada. Como los árboles de expresión son inmutables, se pueden crear árboles nuevos mediante el uso de los componentes de árboles existentes. Esto minimiza la cantidad de memoria necesaria para crear árboles de expresión modificados.

Siguiente: Resumen

Resumen de árboles de expresión

18/03/2020 • 2 minutes to read • Edit Online

Anterior: Traducción de expresiones

En esta serie, se vio cómo se pueden usar *árboles de expresión* para crear programas dinámicos que interpretan el código como datos y crean nueva funcionalidad basada en ese código.

Se pueden examinar los árboles de expresión para entender el propósito de un algoritmo. No se puede examinar solo ese código. Se pueden crear árboles de expresión nuevos que representen las versiones modificadas del código original.

También se pueden usar árboles de expresión para buscar en un algoritmo y traducirlo a otro lenguaje o entorno.

Limitaciones

Hay algunos elementos de lenguaje de C# nuevos que no se traducen correctamente en árboles de expresión. Los árboles de expresión no pueden contener expresiones `await` ni expresiones lambda `async`. Muchas de las características agregadas en la versión 6 de C# no aparecen tal y como se escriben en árboles de expresión. En su lugar, se expondrán las características más recientes de los árboles de expresión en la sintaxis equivalente anterior. Es posible que esta limitación no sea tanta como podría parecer. De hecho, significa que el código que interpreta los árboles de expresión probablemente funcionará igual cuando se introduzcan las nuevas características de lenguaje.

Incluso con estas limitaciones, los árboles de expresión permiten crear algoritmos dinámicos que se basan en la interpretación y la modificación del código que se representa como una estructura de datos. Es una herramienta eficaz y es una de las características del ecosistema .NET que permite que las bibliotecas enriquecidas como Entity Framework realicen lo que hacen.

Interoperabilidad (Guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

La interoperabilidad permite conservar y aprovechar las inversiones existentes en código no administrado. El código que se ejecuta bajo el control de Common Language Runtime (CLR) se denomina *código administrado*, y el código que se ejecuta fuera de CLR se denomina *código no administrado*. COM, COM+, los componentes de C++, los componentes de ActiveX y la API Windows de Microsoft son ejemplos de código no administrado.

.NET habilita la interoperabilidad con código no administrado a través de los servicios de invocación de plataforma, el espacio de nombres System.Runtime.InteropServices, la interoperabilidad de C++ y la interoperabilidad COM.

En esta sección

Información general sobre interoperabilidad
Se describen métodos para habilitar la interoperabilidad entre el código administrado y el código no administrado de C#.

Procedimiento para acceder a objetos de interoperabilidad de Office mediante características de C#
Describe las características introducidas en Visual C# para facilitar la programación de Office.

Procedimiento para usar propiedades indizadas en la programación de interoperabilidad COM
Se describe cómo utilizar las propiedades indizadas para acceder a las propiedades de COM que tienen parámetros.

Procedimiento para usar la invocación de plataforma para reproducir un archivo WAV
Se describe cómo usar los servicios de invocación de plataforma para reproducir un archivo de sonido .wav en el sistema operativo Windows.

Tutorial: Programación de Office
Muestra cómo crear un libro de Excel y un documento de Word que contiene un vínculo al libro.

Clase COM de ejemplo
Muestra cómo exponer una clase de C# como un objeto COM.

Especificación del lenguaje C#

Para obtener más información, consulte la sección Conceptos básicos de Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Marshal.ReleaseComObject
- Guía de programación de C#
- Interoperar con código no administrado
- Tutorial: Programación de Office

Documentación del código con comentarios XML

18/03/2020 • 37 minutes to read • Edit Online

Los comentarios de documentación XML son un tipo especial de comentarios que se agregan encima de la definición de un tipo o un miembro definido por el usuario. Son especiales porque los puede procesar el compilador para generar un archivo de documentación XML en tiempo de compilación. El archivo XML generado por el compilador se puede distribuir junto con el ensamblado .NET de modo que Visual Studio y otros IDE puedan usar IntelliSense para mostrar información rápida sobre los tipos o los miembros. Además, el archivo XML se puede ejecutar mediante herramientas como DocFX y Sandcastle para generar sitios web de referencia de API.

El compilador omite los comentarios de documentación XML, igual que los demás comentarios.

Para generar el archivo XML en tiempo de compilación, realice una de las siguientes acciones:

- Si está desarrollando una aplicación con .NET Core desde la línea de comandos, puede agregar un elemento `GenerateDocumentationFile` a la sección `<PropertyGroup>` de su archivo de proyecto .csproj. También puede especificar la ruta de acceso al archivo de documentación directamente mediante el elemento `DocumentationFile`. En el siguiente ejemplo se genera un archivo XML en el directorio del proyecto con el mismo nombre de archivo raíz que el ensamblado:

  ```
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  ```

  Es equivalente a la siguiente:

  ```
  <DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</DocumentationFile>
  ```

- Si está desarrollando una aplicación mediante Visual Studio, haga clic con el botón derecho en el proyecto y seleccione **Propiedades**. En el cuadro de diálogo de propiedades, seleccione la pestaña **Compilar** y active **Archivo de documentación XML**. También puede cambiar la ubicación en la que el compilador escribe el archivo.

- Si está compilando una aplicación de .NET Framework desde la línea de comandos, agregue la opción del compilador -doc al compilar.

Los comentarios de documentación XML usan tres barras diagonales (`///`) y un cuerpo de comentario con formato XML. Por ejemplo:

```
/// <summary>
/// This class does something.
/// </summary>
public class SomeClass
{
}
```

Tutorial

Vamos a documentar una biblioteca matemática muy básica para que a los nuevos desarrolladores les resulte más fácil comprenderla y contribuir, y a los desarrolladores de otras empresas usarla.

Este es el código de la biblioteca matemática simple:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
public class Math
{
    // Adds two integers and returns the result
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

La biblioteca de ejemplo es compatible con cuatro operaciones aritméticas principales (`add`, `subtract`, `multiply` y `divide`) en tipos de datos `int` y `double`.

Ahora le interesa crear un documento de referencia de API desde el código para los desarrolladores de otras empresas que usan la biblioteca, pero no tienen acceso al código fuente. Como se mencionó anteriormente, las etiquetas de documentación XML pueden utilizarse para lograr esto. Ahora recibirá una introducción a las etiquetas XML estándares que admite el compilador de C#.

<summary>

La etiqueta `<summary>` agrega información breve sobre un tipo o miembro. Vamos a demostrar su uso agregándola a la definición de clase `Math` y al primer método `Add`. No dude en aplicarla al resto del código.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

La etiqueta `<summary>` es importante y se recomienda incluirla, dado que su contenido es la principal fuente de información sobre el tipo o el miembro en IntelliSense o en un documento de referencia de API.

<remarks>

La etiqueta `<remarks>` complementa la información sobre los tipos o los miembros que proporciona la etiqueta `<summary>`. En este ejemplo, solo la agregará a la clase.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// </remarks>
public class Math
{
}
```

<returns>

La etiqueta `<returns>` describe el valor devuelto de una declaración de método. Al igual que antes, en el ejemplo siguiente se muestra la etiqueta `<returns>` en el primer método `Add`. Puede hacer lo mismo en otros métodos.

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

<value>

La etiqueta `<value>` es similar a la etiqueta `<returns>`, salvo que se usa para propiedades. Supongamos que su biblioteca `Math` tenía una propiedad estática denominada `PI`. A continuación le mostramos cómo usaría esta etiqueta:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// These operations can be performed on both integers and doubles
/// </remarks>
public class Math
{
    /// <value>Gets the value of PI.</value>
    public static double PI { get; }
}
```

<example>

La etiqueta `<example>` se usa para incluir un ejemplo en la documentación XML. Esto implica usar la etiqueta secundaria `<code>`.

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Add(4, 5);
/// if (c > 10)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

La etiqueta `code` conserva los saltos de línea y la sangría en los ejemplos más largos.

<para>

La etiqueta `<para>` se usa para dar formato al contenido dentro de la etiqueta primaria. `<para>` suele usarse dentro de una etiqueta, como `<remarks>` o `<returns>`, para dividir el texto en párrafos. Puede dar formato al contenido de la etiqueta `<remarks>` de su definición de clase.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
}
```

<c>

También en el ámbito del formato, puede usar la etiqueta `<c>` para marcar una parte del texto como código. Es como la etiqueta `<code>`, pero insertada. Es útil si quiere mostrar un ejemplo de código rápido como parte del contenido de la etiqueta. Vamos a actualizar la documentación de la clase `Math`.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
}
```

<exception>

Mediante el uso de la etiqueta `<exception>`, permite que los desarrolladores sepan que un método puede producir excepciones específicas. Si examina su biblioteca `Math`, verá que los dos métodos `Add` producen una excepción si se cumple una determinada condición. En cambio, no resulta tan obvio que el método entero `Divide` también produce una si el parámetro `b` es cero. Vamos a agregar documentación de la excepción a este método.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    public static double Add(double a, double b)
    {
```

```
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Divides an integer by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two integers.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    /// <summary>
    /// Divides a double by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two doubles.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

El atributo `cref` representa una referencia a una excepción que está disponible desde el entorno de compilación actual. Puede ser cualquier tipo definido en el proyecto o un ensamblado de referencia. El compilador emitirá una advertencia si su valor no puede resolverse.

<see>

La etiqueta `<see>` le permite crear un vínculo interactivo a una página de documentación para otro elemento de código. En el siguiente ejemplo, crearemos un vínculo interactivo entre los dos métodos `Add`.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

`cref` es un atributo **necesario** que representa una referencia a un tipo o a su miembro que está disponible desde el entorno de compilación actual. Puede ser cualquier tipo definido en el proyecto o un ensamblado de referencia.

<seealso>

La etiqueta `<seealso>` se usa de la misma manera que la etiqueta `<see>`. La única diferencia es que su contenido se suele colocar en una sección "Vea también". Aquí vamos a agregar una etiqueta `seealso` en el método entero `Add` para hacer referencia a otros métodos de la clase que aceptan parámetros enteros:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

El atributo `cref` representa una referencia a un tipo o a su miembro que está disponible desde el entorno de compilación actual. Puede ser cualquier tipo definido en el proyecto o un ensamblado de referencia.

<param>

La etiqueta `<param>` se usa para describir los parámetros de un método. Este es un ejemplo sobre el método doble `Add`: El parámetro que describe la etiqueta se especifica en el atributo **necesario** `name`.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<typeparam>

La etiqueta `<typeparam>` se usa igual que la etiqueta `<param>`, pero sirve para que las declaraciones de método o tipo genérico describan un parámetro genérico. Agregue un método genérico rápido a su clase `Math` para comprobar si una cantidad es mayor que otra.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}
```

<paramref>

Puede darse la situación de que esté describiendo lo que hace un método en una etiqueta `<summary>` y le interese hacer referencia a un parámetro. La etiqueta `<paramref>` es perfecta para esto. Vamos a actualizar el resumen del método doble `Add`. Igual que en la etiqueta `<param>`, el nombre del parámetro se especifica en el atributo

necesario `name`.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<typeparamref>

La etiqueta `<typeparamref>` se usa igual que la etiqueta `<paramref>`, pero sirve para que las declaraciones de método o tipo genérico describan un parámetro genérico. Puede usar el mismo método genérico que ha creado previamente.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable <typeparamref name="T"/> is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}
```

<list>

La etiqueta `<list>` se usa para dar formato a la información de la documentación en una lista ordenada, una lista

sin ordenar o una tabla. Cree una lista sin ordenar con todas las operaciones matemáticas que admita su biblioteca `Math`.

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
public class Math
{
}
```

Para crear una lista ordenada o una tabla, cambie el atributo `type` a `number` o `table`, respectivamente.

<inheritdoc>

Puede usar la etiqueta `<inheritdoc>` para heredar comentarios XML de clases base, interfaces y métodos similares. Esto elimina la acción de copiar y pegar no deseada de comentarios XML duplicados y mantiene los comentarios XML sincronizados automáticamente.

```
/*
    The IMath interface
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// This is the IMath interface.
/// </summary>
public interface IMath
{
}

/// <inheritdoc/>
public class Math : IMath
{
}
```

En resumen

Si ha seguido este tutorial y ha aplicado las etiquetas al código en los casos en que era necesario, el código debe ser ahora similar al siguiente:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"/> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <summary>
```

```
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Add(4.5, 5.4);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(int, int)"/> to add integers.
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <summary>
    /// Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The difference between two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Subtract(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <summary>
    /// Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
result.
    /// </summary>
    /// <returns>
    /// The difference between two doubles.
    /// </returns>
    /// <example>
    /// <code>
```

```
    /// double c = Math.Subtract(4.5, 5.4);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Subtract(int, int)"/> to subtract integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <summary>
    /// Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The product of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Multiply(4, 5);
    /// if (c > 100)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Divide(int, int)"/>
    /// <param name="a">An integer.</param>
    /// <param name="b">An integer.</param>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <summary>
    /// Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The product of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Multiply(4.5, 5.4);
    /// if (c > 100.0)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// See <see cref="Math.Multiply(int, int)"/> to multiply integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Divide(double, double)"/>
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Multiply(double a, double b)
    {
```

```
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <summary>
    /// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The quotient of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Divide(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(double, double)"/> to divide doubles.
    /// <seealso cref="Math.Add(int, int)"/>
    /// <seealso cref="Math.Subtract(int, int)"/>
    /// <seealso cref="Math.Multiply(int, int)"/>
    /// <param name="a">An integer dividend.</param>
    /// <param name="b">An integer divisor.</param>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <summary>
    /// Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The quotient of two doubles.
    /// </returns>
    /// <example>
    /// <code>
    /// double c = Math.Divide(4.5, 5.4);
    /// if (c > 1.0)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(int, int)"/> to divide integers.
    /// <seealso cref="Math.Add(double, double)"/>
    /// <seealso cref="Math.Subtract(double, double)"/>
    /// <seealso cref="Math.Multiply(double, double)"/>
    /// <param name="a">A double precision dividend.</param>
    /// <param name="b">A double precision divisor.</param>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

Desde su código, puede generar un sitio web de documentación detallada completo con referencias cruzadas interactivas. Pero se enfrenta a otro problema: su código se ha vuelto difícil de leer. Es una pesadilla para los desarrolladores que quieran contribuir a este código, ya que hay mucha información que examinar. Afortunadamente, hay una etiqueta XML que le ayudará a resolverlo:

<include>

La etiqueta `<include>` le permite hacer referencia a los comentarios de un archivo XML independiente que describen los tipos y los miembros del código fuente, en vez de colocar los comentarios de documentación directamente en el archivo de código fuente.

Ahora vamos a mover todas las etiquetas XML a un archivo XML independiente denominado `docs.xml`. Puede ponerle al archivo el nombre que quiera.

```
<docs>
    <members name="math">
        <Math>
            <summary>
            The main <c>Math</c> class.
            Contains all methods for performing basic math functions.
            </summary>
            <remarks>
            <para>This class can add, subtract, multiply and divide.</para>
            <para>These operations can be performed on both integers and doubles.</para>
            </remarks>
        </Math>
        <AddInt>
            <summary>
            Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The sum of two integers.
            </returns>
            <example>
            <code>
            int c = Math.Add(4, 5);
            if (c > 10)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            <exception cref="System.OverflowException">Thrown when one parameter is max
            and the other is greater than 0.</exception>
            See <see cref="Math.Add(double, double)"/> to add doubles.
            <seealso cref="Math.Subtract(int, int)"/>
            <seealso cref="Math.Multiply(int, int)"/>
            <seealso cref="Math.Divide(int, int)"/>
            <param name="a">An integer.</param>
            <param name="b">An integer.</param>
        </AddInt>
        <AddDouble>
            <summary>
            Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The sum of two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Add(4.5, 5.4);
            if (c > 10)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            <exception cref="System.OverflowException">Thrown when one parameter is max
            and the other is greater than 0.</exception>
            See <see cref="Math.Add(int, int)"/> to add integers.
            <seealso cref="Math.Subtract(double, double)"/>
```

```
            <seealso cref="Math.Multiply(double, double)"/>
            <seealso cref="Math.Divide(double, double)"/>
            <param name="a">A double precision number.</param>
            <param name="b">A double precision number.</param>
        </AddDouble>
        <SubtractInt>
            <summary>
            Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
            </summary>
            <returns>
            The difference between two integers.
            </returns>
            <example>
            <code>
            int c = Math.Subtract(4, 5);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
            <seealso cref="Math.Add(int, int)"/>
            <seealso cref="Math.Multiply(int, int)"/>
            <seealso cref="Math.Divide(int, int)"/>
            <param name="a">An integer.</param>
            <param name="b">An integer.</param>
        </SubtractInt>
        <SubtractDouble>
            <summary>
            Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
result.
            </summary>
            <returns>
            The difference between two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Subtract(4.5, 5.4);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Subtract(int, int)"/> to subtract integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Multiply(double, double)"/>
            <seealso cref="Math.Divide(double, double)"/>
            <param name="a">A double precision number.</param>
            <param name="b">A double precision number.</param>
        </SubtractDouble>
        <MultiplyInt>
            <summary>
            Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The product of two integers.
            </returns>
            <example>
            <code>
            int c = Math.Multiply(4, 5);
            if (c > 100)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
```

```
            <seealso cref="Math.Add(int, int)"/>
            <seealso cref="Math.Subtract(int, int)"/>
            <seealso cref="Math.Divide(int, int)"/>
            <param name="a">An integer.</param>
            <param name="b">An integer.</param>
        </MultiplyInt>
        <MultiplyDouble>
            <summary>
            Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
            </summary>
            <returns>
            The product of two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Multiply(4.5, 5.4);
            if (c > 100.0)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            See <see cref="Math.Multiply(int, int)"/> to multiply integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Subtract(double, double)"/>
            <seealso cref="Math.Divide(double, double)"/>
            <param name="a">A double precision number.</param>
            <param name="b">A double precision number.</param>
        </MultiplyDouble>
        <DivideInt>
            <summary>
            Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the
result.
            </summary>
            <returns>
            The quotient of two integers.
            </returns>
            <example>
            <code>
            int c = Math.Divide(4, 5);
            if (c > 1)
            {
                Console.WriteLine(c);
            }
            </code>
            </example>
            <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
            See <see cref="Math.Divide(double, double)"/> to divide doubles.
            <seealso cref="Math.Add(int, int)"/>
            <seealso cref="Math.Subtract(int, int)"/>
            <seealso cref="Math.Multiply(int, int)"/>
            <param name="a">An integer dividend.</param>
            <param name="b">An integer divisor.</param>
        </DivideInt>
        <DivideDouble>
            <summary>
            Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the
result.
            </summary>
            <returns>
            The quotient of two doubles.
            </returns>
            <example>
            <code>
            double c = Math.Divide(4.5, 5.4);
            if (c > 1.0)
            {
                Console.WriteLine(c);
```

```
            }
            </code>
            </example>
            <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
            See <see cref="Math.Divide(int, int)"/> to divide integers.
            <seealso cref="Math.Add(double, double)"/>
            <seealso cref="Math.Subtract(double, double)"/>
            <seealso cref="Math.Multiply(double, double)"/>
            <param name="a">A double precision dividend.</param>
            <param name="b">A double precision divisor.</param>
        </DivideDouble>
    </members>
</docs>
```

En el XML anterior, los comentarios de documentación de cada miembro aparecen directamente dentro de una etiqueta cuyo nombre indica lo que hacen, pero puede elegir su propia estrategia. Ahora que tiene los comentarios XML en un archivo independiente, veamos cómo se puede hacer más legible el código mediante la etiqueta `<include>`:

```
/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <include file='docs.xml' path='docs/members[@name="math"]/Math/*'/>
public class Math
{
    // Adds two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddInt/*'/>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddDouble/*'/>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractInt/*'/>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractDouble/*'/>
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
```

```
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyInt/*'/>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyDouble/*'/>
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideInt/*'/>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideDouble/*'/>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}
```

Aquí lo tiene: el código vuelve a ser legible y no se ha perdido ninguna información de documentación.

El atributo `file` representa el nombre del archivo XML que contiene la documentación.

El atributo `path` representa una consulta XPath para el `tag name` presente en el `file` especificado.

El atributo `name` representa el especificador de nombre en la etiqueta que precede a los comentarios.

El atributo `id`, que se puede usar en lugar de `name`, representa el identificador de la etiqueta que precede a los comentarios.

Etiquetas definidas por el usuario

Todas las etiquetas descritas anteriormente son las que reconoce el compilador de C#, pero el usuario puede definir sus propias etiquetas. Las herramientas como Sandcastle proporcionan compatibilidad con etiquetas adicionales como <event> y <note>, e incluso permiten documentar espacios de nombres. También se pueden usar herramientas de generación de documentación internas o personalizadas con las etiquetas estándar y se admiten varios formatos de salida, de HTML a PDF.

Recomendaciones

Se recomienda documentar código por diversos motivos. A continuación se muestran algunos procedimientos recomendados, escenarios generales de casos de uso y conceptos que debe conocer al usar etiquetas de documentación XML en el código de C#.

- Por motivos de coherencia, se deben documentar todos los tipos públicamente visibles y sus miembros. Si debe hacerlo, hágalo en todos los elementos.
- Los miembros privados también se pueden documentar mediante comentarios XML, pero esto expone el funcionamiento interno (potencialmente confidencial) de la biblioteca.
- Como mínimo, los tipos y sus miembros deben tener una etiqueta `<summary>`, ya que su contenido es necesario para IntelliSense.
- El texto de la documentación se debe escribir con frases completas que terminen en punto.
- Las clases parciales son totalmente compatibles y la información de documentación se concatenará en una

única entrada para ese tipo.

- El compilador comprueba la sintaxis de las etiquetas `<exception>`, `<include>`, `<param>`, `<see>`, `<seealso>` y `<typeparam>`.
- El compilador valida los parámetros que contienen rutas de acceso de archivo y referencias a otras partes del código.

Vea también

- Comentarios de documentación XML (Guía de programación de C#)
- Etiquetas recomendadas para comentarios de documentación (Guía de programación de C#)

Control de versiones en C#

27/04/2020 • 11 minutes to read • Edit Online

En este tutorial, obtendrá información sobre qué significa el control de versiones en .NET. También obtendrá información sobre los factores que deben tenerse en cuenta para controlar las versiones de su biblioteca así como para actualizar a una versión nueva de esta.

Creación de bibliotecas

Como desarrollador que ha creado bibliotecas de .NET para uso público, probablemente se ha encontrado en situaciones en las que tiene que implementar nuevas actualizaciones. Cómo realizar este proceso es muy importante, ya que necesita asegurarse de que existe una transición sin problemas del código existente a la versión nueva de su biblioteca. Aquí se muestran algunos aspectos para tener en cuenta a la hora de crear una versión nueva:

Control de versiones semántico

Control de versiones semántico (SemVer para abreviar) es una convención de nomenclatura que se aplica a las versiones de su biblioteca para indicar eventos importantes específicos. De manera ideal, la información de la versión que proporciona a la biblioteca debe ayudar a los desarrolladores a determinar la compatibilidad con sus proyectos que usan versiones anteriores de la misma biblioteca.

El enfoque más sencillo de SemVer es el formato de 3 componentes `MAJOR.MINOR.PATCH`, donde:

- `MAJOR` se incrementa cuando realiza cambios de API incompatibles
- `MINOR` se incrementa cuando agrega funciones de manera compatible con versiones anteriores
- `PATCH` se incrementa cuando realiza correcciones de errores compatibles con versiones anteriores

También existen maneras de especificar otros escenarios como versiones preliminares, etc. al aplicar información de la versión a su biblioteca .NET.

Compatibilidad con versiones anteriores

A medida que presente versiones nuevas de su biblioteca, la compatibilidad con las versiones anteriores será probablemente una de sus mayores preocupaciones. Una versión nueva de su biblioteca es compatible en su origen con una versión anterior si el código que depende de la versión anterior, puede, cuando se vuelve a compilar, trabajar con la versión nueva. Una versión nueva de su biblioteca tiene compatibilidad binaria si una aplicación que dependía de la versión anterior, puede, sin que se vuelva a compilar, trabajar con la versión nueva.

Aquí se muestran algunos aspectos a tener en cuenta al intentar mantener la compatibilidad con versiones anteriores de su biblioteca:

- Métodos virtuales: cuando hace que un método virtual sea no virtual en la versión nueva, significa que los proyectos que reemplacen ese método tendrán que actualizarse. Esto es un cambio brusco enorme y se desaconseja totalmente.
- Firmas de método: cuando actualizar un comportamiento del método requiere que también se cambie su firma, en su lugar se debe crear una sobrecarga de manera que el código que llama a ese método siga funcionando. Siempre puede manipular la firma del método anterior para llamar a la firma del método nuevo, de manera que la implementación siga siendo coherente.
- Atributo obsoleto: puede usar este atributo en el código para especificar clases o miembros de clases que han quedado obsoletos y que probablemente se quiten en versiones futuras. Esto garantiza que los desarrolladores que usen su biblioteca estén mejor preparados para los cambios bruscos.

- Argumentos de método opcionales: cuando hace que los argumentos de método opcionales anteriores sean obligatorios o cambien su valor predeterminado, se tendrá que actualizar todo el código que no proporcione esos argumentos.

> **NOTE**
> Hacer que los argumentos obligatorios sean opcionales debe tener un efecto muy pequeño, especialmente si no cambia el comportamiento del método.

Cuánto más facilite la actualización a la nueva versión de la biblioteca a sus usuarios, más rápidamente la actualizarán.

Archivo de configuración de aplicación

Como desarrollador de .NET, existe una posibilidad muy alta de que haya encontrado el archivo `app.config` en la mayoría de tipos de proyecto. Este sencillo archivo de configuración puede hacer mucho por mejorar la implementación de las actualizaciones nuevas. Generalmente, debe diseñar sus bibliotecas de tal manera que la información que es probable que cambie regularmente se almacene en el archivo `app.config`, de esta manera, cuando dicha información se actualice, el archivo de configuración de las versiones anteriores solo necesita reemplazarse por el nuevo sin necesidad de volver a compilar la biblioteca.

Consumo de bibliotecas

Como desarrollador que consume bibliotecas .NET creadas por otros desarrolladores, es probable que sea consciente de que una nueva versión de una biblioteca puede que no sea completamente compatible con su proyecto y, a menudo, puede que tenga que actualizar su código para trabajar con esos cambios.

Por suerte, C# y el ecosistema de .NET incluyen características y técnicas que nos permiten actualizar fácilmente nuestra aplicación para que funcione con las versiones nuevas de bibliotecas que pueden presentar cambios bruscos.

Redirección de enlace de ensamblados

Puede usar el archivo *app.config* para actualizar la versión de una biblioteca que use su aplicación. Al agregar lo que se denomina una *redirección de enlace*, se puede usar la nueva versión de la biblioteca sin tener que volver a compilar la aplicación. En el siguiente ejemplo se muestra cómo actualizaría el archivo *app.config* de la aplicación para usar la versión de revisión `1.0.1` de `ReferencedLibrary`, en lugar de la versión `1.0.0` con la que se ha compilado originalmente.

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> **NOTE**
> Este enfoque solo funcionará si la versión nueva de `ReferencedLibrary` tiene compatibilidad binaria con su aplicación. Vea la sección anterior Compatibilidad con versiones anteriores para ver los cambios que debe tener en cuenta al determinar la compatibilidad.

new

Use el modificador `new` para ocultar los miembros heredados de una clase base. Esta es una manera en la que las clases derivadas pueden responder a las actualizaciones en clases base.

Considere el ejemplo siguiente:

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

Salida

```
A base method
A derived method
```

En el ejemplo anterior puede ver cómo `DerivedClass` oculta el método `MyMethod` presente en `BaseClass`. Esto significa que cuando una clase base en la versión nueva de una biblioteca agrega un miembro que ya existe en su clase derivada, simplemente puede usar el modificador `new` en su miembro de clase derivada para ocultar el miembro de clase base.

Cuando no se especifica ningún modificador `new`, una clase derivada ocultará de manera predeterminada los miembros en conflicto de una clase base, aunque se generará una advertencia del compilador, el código se compilará. Esto significa que agregar simplemente miembros nuevos a una clase existente, hace que la versión nueva de su biblioteca tenga compatibilidad binaria y de origen con el código que depende de ella.

override

El modificador `override` significa que una implementación derivada extiende la implementación de un miembro de clase base en lugar de ocultarlo. El miembro de clase base necesita que se le aplique el modificador `virtual`.

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

Salida

```
Base Method One: Method One
Derived Method One: Derived Method One
```

El modificador `override` se evalúa en tiempo de compilación y el compilador producirá un error si no encuentra un miembro virtual que reemplazar.

Su conocimiento de las técnicas que se han tratado y su comprensión de las situaciones en las cuales usarlas, contribuirá en gran medida a facilitar la transición entre las versiones de una biblioteca.

Procedimientos (C#)

07/05/2020 • 7 minutes to read • Edit Online

En la sección de procedimientos de la guía de C# puede encontrar respuestas rápidas a preguntas frecuentes. En algunos casos, los artículos pueden mostrarse en varias secciones. Hemos querido que sean fáciles de encontrar en diferentes rutas de búsqueda.

Conceptos generales de C#

Hay varios trucos y sugerencias que son habituales entre los desarrolladores de C#:

- Inicialice los objetos usando un inicializador de objeto.
- Obtenga información sobre las diferencias entre pasar una estructura o una clase a un método.
- Use la sobrecarga de operadores.
- Implemente e invoque un método de extensión personalizado.
- Incluso los programadores de C# es posible que quieran usar el espacio de nombres `My` de Visual Basic.
- Cree un nuevo método para un tipo `enum` mediante métodos de extensión.

Miembros de clase y estructura

Cree clases y estructuras para implementar su programa. Estas técnicas suelen usarse al escribir clases o estructuras.

- Declare propiedades de implementación automática.
- Declare y use propiedades de lectura y escritura.
- Defina las constantes.
- Invalide el método `ToString` para proporcionar la salida de la cadena.
- Defina las propiedades abstractas.
- Use las características de documentación XML para documentar el código.
- Implemente explícitamente miembros de interfaz para que su interfaz pública sea concisa.
- Implemente explícitamente miembros de dos interfaces.

Trabajar con colecciones

Estos artículos le ayudarán a trabajar con colecciones de datos.

- Inicialice un diccionario con un inicializador de colección.

Trabajo con cadenas

Las cadenas son el tipo de datos básico que se usa para mostrar o manipular texto. En estos artículos se muestran prácticas habituales con cadenas.

- Compare cadenas.
- Modifique el contenido de una cadena.
- Determine si una cadena representa un número.
- Use `String.Split` para separar cadenas.
- Combine varias cadenas en una.
- Busque texto en una cadena.

Conversión entre tipos

Puede que deba convertir un objeto a otro tipo.

- Determine si una cadena representa un número.
- Realice conversiones entre cadenas que representen números hexadecimales y el número.
- Convierta una cadena en un valor `DateTime`.
- Convierta una matriz de bytes en un valor int.
- Convierta una cadena en un número.
- Use la coincidencia de patrones y los operadores `as` y `is` para una conversión segura a otro tipo.
- Definición de las conversiones de tipos personalizadas.
- Determine si un tipo es un tipo de valor que acepta valores NULL.
- Realice conversiones entre tipos de valores que aceptan valores NULL y tipos que no.

Comparaciones de igualdad y ordenación

Puede crear tipos que definan sus propias reglas para la igualdad o que definan una ordenación natural entre los objetos de ese tipo.

- Pruebe la igualdad basada en referencias.
- Defina la igualdad basada en valores de un tipo.

Control de excepciones

Los programas de .NET informan de que un método no se ha ejecutado correctamente y de que se han generado excepciones. En estos artículos aprenderá a trabajar con excepciones.

- Controle excepciones mediante `try` y `catch`.
- Limpie recursos con cláusulas `finally`.
- Recupere excepciones no conformes a CLS (Common Language Specification).

Delegados y eventos

Los delegados y los eventos proporcionan capacidad para las estrategias que implican bloques de código sin una conexión directa.

- Declare y use delegados, y cree instancias de estos.
- Combine delegados multidifusión.

Los eventos son un mecanismo para publicar notificaciones o suscribirse a ellas.

- Suscríbase a eventos y cancele la suscripción a estos.
- Implemente eventos declarados en interfaces.
- Garantice la conformidad a las directrices de .NET Framework cuando el código publica los eventos.
- Genere eventos definidos en las clases base a partir de clases derivadas.
- Implemente descriptores de acceso de eventos personalizados.

Prácticas de LINQ

LINQ permite escribir código para consultar cualquier origen de datos que admita su patrón de expresión de consultas. Estos artículos le ayudarán a comprender el patrón y a trabajar con orígenes de datos diferentes.

- Consulte una colección.
- Use expresiones lambda en una consulta.

- Use `var` en expresiones de consulta.
- Devuelva subconjuntos de propiedades de elementos de una consulta.
- Escriba consultas con filtrado complejo.
- Ordene los elementos de un origen de datos.
- Ordene elementos en varias claves.
- Controle el tipo de una proyección.
- Cuente las repeticiones de un valor en una secuencia de origen.
- Calcule valores intermedios.
- Combine datos de varios orígenes.
- Encuentre la diferencia de conjuntos entre dos secuencias.
- Depure los resultados vacíos de una consulta.
- Agregue métodos personalizados para las consultas de LINQ.

Varios subprocesos y procesamiento asincrónico

Los programas modernos suelen usar operaciones asincrónicas. Estos artículos le ayudarán a aprender a usar estas técnicas.

- Mejore el rendimiento asíncrono usando `System.Threading.Tasks.Task.WhenAll`.
- Realice varias solicitudes web en paralelo usando `async` y `await`.
- Use un grupo de subprocesos.

Argumentos de línea de comandos para el programa

Normalmente, los programas de C# tienen argumentos de línea de comandos. En estos artículos se explica cómo obtener acceso a los argumentos de línea de comandos, además de cómo procesarlos.

- Recupere todos los argumentos de línea de comandos con `for`.

Procedimiento para analizar cadenas mediante String.Split en C#

29/06/2020 • 4 minutes to read • Edit Online

El método String.Split crea una matriz de subcadenas mediante la división de la cadena de entrada en función de uno o varios delimitadores. Suele ser la manera más fácil de separar una cadena en límites de palabras. También sirve para dividir las cadenas en otras cadenas o caracteres específicos.

> **NOTE**
>
> Los ejemplos de C# de este artículo se ejecutan en el ejecutor de código en línea y área de juegos de Try.NET. Haga clic en el botón **Ejecutar** para ejecutar un ejemplo en una ventana interactiva. Una vez que se ejecuta el código, puede modificar y ejecutar el código modificado si vuelve a hacer clic en **Ejecutar**. El código modificado se ejecuta en la ventana interactiva o, si se produce un error en la compilación, en la ventana interactiva se muestran todos los mensajes de error del compilador de C#.

Este código divide una frase común en una matriz de cadenas para cada palabra.

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Todas las instancias de un carácter separador generan un valor en la matriz devuelta. Los caracteres separadores consecutivos generan la cadena vacía como un valor en la matriz devuelta. Puede ver cómo se crea una cadena vacía en el ejemplo siguiente, en el que se usa el carácter de espacio como separador.

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Este comportamiento facilita formatos como los de los archivos de valores separados por comas (CSV) que representan datos tabulares. Las comas consecutivas representan una columna en blanco.

Puede pasar un parámetro StringSplitOptions.RemoveEmptyEntries opcional para excluir cualquier cadena vacía en la matriz devuelta. Para un procesamiento más complicado de la colección devuelta, puede usar LINQ para manipular la secuencia de resultados.

String.Split puede usar varios caracteres separadores. En este ejemplo se usan espacios, comas, puntos, dos puntos y tabulaciones; están incluidos todos en una matriz que contiene estos caracteres de separación y que se pasa a Split. En el bucle al final del código se muestra cada una de las palabras de la matriz devuelta.

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

Las instancias consecutivas de cualquier separador generan la cadena vacía en la matriz de salida:

```
char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

String.Split puede tomar una matriz de cadenas (secuencias de caracteres que actúan como separadores para analizar la cadena de destino, en lugar de caracteres individuales).

```
string[] separatingStrings = { "<<", "..." };

string text = "one<<two......three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings, System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

Vea también

- Guía de programación de C#
- Cadenas
- Expresiones regulares de .NET

Procedimiento para concatenar varias cadenas (Guía de C#)

29/06/2020 • 6 minutes to read • Edit Online

Concatenación es el proceso de anexar una cadena al final de otra cadena. Las cadenas se concatenan con el operador `+`. En el caso de los literales y las constantes de cadena, la concatenación se produce en tiempo de compilación, y no en tiempo de ejecución. En cambio, para las variables de cadena, la concatenación solo se produce en tiempo de ejecución.

> **NOTE**
>
> Los ejemplos de C# de este artículo se ejecutan en el ejecutor de código en línea y área de juegos de Try.NET. Haga clic en el botón **Ejecutar** para ejecutar un ejemplo en una ventana interactiva. Una vez que se ejecuta el código, puede modificar y ejecutar el código modificado si vuelve a hacer clic en **Ejecutar**. El código modificado se ejecuta en la ventana interactiva o, si se produce un error en la compilación, en la ventana interactiva se muestran todos los mensajes de error del compilador de C#.

En el ejemplo siguiente se usa la concatenación para dividir un literal de cadena larga en cadenas más pequeñas para mejorar la legibilidad del código fuente. Estas partes se concatenan en una sola cadena en tiempo de compilación. No existe ningún costo de rendimiento en tiempo de ejecución independientemente del número de cadenas de que se trate.

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
"have not been well integrated. Programmers work in C# or Visual Basic " +
"and also in SQL or XQuery. On the one side are concepts such as classes, " +
"objects, fields, inheritance, and .NET Framework APIs. On the other side " +
"are tables, columns, rows, nodes, and separate languages for dealing with " +
"them. Data types often require translation between the two worlds; there are " +
"different standard functions. Because the object world has no notion of query, a " +
"query can only be represented as a string without compile-time type checking or " +
"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
"objects in memory is often tedious and error-prone.";

System.Console.WriteLine(text);
```

Para concatenar variables de cadena, puede usar los operadores `+` o `+=`, la interpolación de cadena o los métodos String.Format, String.Concat, String.Join o StringBuilder.Append. El operador `+` es sencillo de usar y genera un código intuitivo. Aunque use varios operadores `+` en una instrucción, el contenido de la cadena se copiará solo una vez. En el código siguiente se muestran ejemplos del uso de los operadores `+` y `+=` para concatenar cadenas:

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```

En algunas expresiones, es más fácil concatenar cadenas mediante la interpolación de cadena, como se muestra en este código:

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

> **NOTE**
>
> En operaciones de concatenación de cadenas, el compilador de C# trata una cadena NULL igual que una cadena vacía.

Otro método para concatenar cadenas es String.Format. Este método funciona bien para compilar una cadena a partir de un número reducido de cadenas de componente.

En otros casos, puede combinar cadenas en un bucle, donde no sabe cuántas cadenas de origen se combinan, y el número real de cadenas de origen puede ser elevado. La clase StringBuilder se diseñó para estos escenarios. El código siguiente usa el método Append de la clase StringBuilder para concatenar cadenas.

```
// Use StringBuilder for concatenation in tight loops.
var sb = new System.Text.StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
System.Console.WriteLine(sb.ToString());
```

Puede obtener más información sobre las razones para elegir la concatenación de cadenas o sobre la clase `StringBuilder`.

Otra opción para combinar cadenas a partir de una colección consiste en usar el método String.Concat. Use el método String.Join si las cadenas de origen se deben separar con un delimitador. El código siguiente combina una matriz de palabras usando ambos métodos:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var unreadablePhrase = string.Concat(words);
System.Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

Por último, puede usar LINQ y el método Enumerable.Aggregate para combinar cadenas a partir de una colección. Este método combina las cadenas de origen mediante una expresión lambda. La expresión lambda realiza el trabajo de agregar cada cadena a la acumulación existente. En el ejemplo siguiente se combina una matriz de palabras mediante la adición de un espacio entre cada palabra de la matriz:

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) =>$"{partialPhrase} {word}");
System.Console.WriteLine(phrase);
```

Vea también

- String
- StringBuilder
- Guía de programación de C#
- Cadenas

Análisis de cadenas de fecha y hora en .NET

29/06/2020 • 13 minutes to read • Edit Online

El análisis de cadenas para convertirlas en objetos DateTime requiere que se especifique información sobre cómo se representan las fechas y horas en forma de texto. Las distintas referencias culturales usan distintos órdenes de día, mes y año. Algunas representaciones horarias usan el reloj de 24 horas y otras especifican "a. m." y "p. m.". Algunas aplicaciones solo necesitan la fecha. Otras solo necesitan la hora. Pero otras necesitan especificar la fecha y la hora. Los métodos que convierten cadenas a objeto DateTime permiten especificar información detallada sobre los formatos esperados y los elementos de fecha y hora que precisa la aplicación. Hay tres subtareas para convertir correctamente el texto en un objeto DateTime:

1. Debe especificarse el formato esperado del texto que representa una fecha y hora.
2. Es posible especificar la referencia cultural del formato de fecha y hora.
3. Puede especificarse cómo se establecen los componentes que faltan en la representación de texto en la fecha y hora.

Los métodos Parse y TryParse convierten varias representaciones comunes de fecha y hora. Los métodos ParseExact y TryParseExact convierten una representación de cadena que se ajusta al modelo especificado por una cadena de formato de fecha y hora. (Para obtener más información, vea los artículos sobre cadenas con formato de fecha y hora estándar y cadenas con formato de fecha y hora personalizado).

El objeto DateTimeFormatInfo actual proporciona más control sobre la forma en que debe interpretarse el texto como fecha y hora. Las propiedades de un objeto DateTimeFormatInfo describen los separadores de fecha y hora, los nombres de los meses, días y eras, así como el formato de las designaciones "a. m." y "p. m.". La referencia cultural del subproceso actual proporciona un objeto DateTimeFormatInfo que representa la referencia cultural actual. Si quiere una referencia cultural específica o una configuración personalizada, especifique el parámetro IFormatProvider de un método de análisis. Para el parámetro IFormatProvider, especifique un objeto CultureInfo, que representa una referencia cultural, o un objeto DateTimeFormatInfo.

Es posible que al texto que representa una fecha y hora le falte alguna información. Por ejemplo, la mayoría de los usuarios supondría que la fecha "12 de marzo" representa el año actual. Del mismo modo, "Marzo de 2018" representa el mes de marzo del año 2018. El texto que representa la hora a menudo solo incluye horas, minutos y una designación "a. m."/"p. m.". Los métodos de análisis controlan esta información que falta mediante valores predeterminados razonables:

- Cuando solo se indica la hora, la parte de fecha utiliza la fecha actual.
- Cuando solo se indica la fecha, la parte de hora es la medianoche.
- Cuando no se especifica el año en una fecha, se usa el año actual.
- Cuando no se especifica el día del mes, se usa el primero del mes.

Si la fecha se indica en la cadena, debe incluir el mes y uno de los dos valores, el día o el año. Si se indica la hora, debe incluir la hora y los minutos o el designador "a. m."/"p. m.".

Se puede especificar la constante NoCurrentDateDefault para invalidar los valores predeterminados. Cuando se usa esta constante, las propiedades de año, mes o día se establecen en el valor `1`. El último ejemplo con Parse muestra este comportamiento.

Además de un componente de fecha y hora, la representación de cadena de una fecha y hora puede incluir una diferencia horaria que indica cuánto difiere la hora respecto de la hora universal coordinada (UTC). Por ejemplo, la cadena "14/2/2007 5:32:00 -7:00" define una hora que es siete horas anterior a la hora UTC. Si se omite la diferencia horaria de la representación de cadena de una hora, el análisis devuelve un objeto DateTime con su

propiedad Kind establecida en DateTimeKind.Unspecified. Si se especifica la diferencia horaria, el análisis devuelve un objeto DateTime con su propiedad Kind establecida en DateTimeKind.Local y su valor ajustado a la zona horaria local del equipo. Puede modificar este comportamiento usando un valor DateTimeStyles con el método de análisis.

El proveedor de formato también se usa para interpretar una fecha numérica ambigua. No queda claro qué componentes de la fecha representada por la cadena "02/03/04" son el mes, el día y el año. Los componentes se interpretan según el orden de formatos de fecha similares del proveedor de formato.

Parse

En el ejemplo siguiente se muestra el uso del método DateTime.Parse para convertir un objeto `string` en un valor DateTime. En este ejemplo se usa la referencia cultural asociada al subproceso actual. Si el objeto CultureInfo asociado a la referencia cultural actual no puede analizar la cadena de entrada, se produce una excepción FormatException.

> **TIP**
> Todos los ejemplos de C# en este artículo se ejecutan en el explorador. Presione el botón **Ejecutar** para ver el resultado. También puede modificarlos para experimentar.

> **NOTE**
> Estos ejemplos están disponibles en el repositorio de documentos de GitHub para C# y Visual Basic.

```
string dateInput = "Jan 1, 2009";
var parsedDate = DateTime.Parse(dateInput);
Console.WriteLine(parsedDate);
// Displays the following output on a system whose culture is en-US:
//       1/1/2009 00:00:00
```

```
Dim MyString As String = "Jan 1, 2009"
Dim MyDateTime As DateTime = DateTime.Parse(MyString)
Console.WriteLine(MyDateTime)
' Displays the following output on a system whose culture is en-US:
'       1/1/2009 00:00:00
```

Además, puede definir explícitamente la referencia cultural cuyas convenciones de formato se utilizan cuando se analiza una cadena. Especifique uno de los objetos DateTimeFormatInfo estándar devueltos por la propiedad CultureInfo.DateTimeFormat. En el ejemplo siguiente se usa un proveedor de formato para analizar una cadena en alemán como valor DateTime. Crea un objeto CultureInfo que representa la referencia cultural `de-DE`. El objeto `CultureInfo` garantiza el análisis correcto de esta cadena concreta. Esto impide cualquier opción que se encuentre en CurrentCulture de CurrentThread.

```
var cultureInfo = new CultureInfo("de-DE");
string dateString = "12 Juni 2008";
var dateTime = DateTime.Parse(dateString, cultureInfo);
Console.WriteLine(dateTime);
// The example displays the following output:
//       6/12/2008 00:00:00
```

```
Dim MyCultureInfo As New CultureInfo("de-DE")
Dim MyString As String = "12 Juni 2008"
Dim MyDateTime As DateTime = DateTime.Parse(MyString, MyCultureInfo)
Console.WriteLine(MyDateTime)
' The example displays the following output:
'       6/12/2008 00:00:00
```

Pero, aunque puede usar sobrecargas del método Parse para especificar proveedores de formato personalizados, el método no admite el análisis de formatos no estándar. Para analizar una fecha y hora expresadas en un formato no estándar, use en su lugar el método ParseExact.

En el ejemplo siguiente se usa la enumeración DateTimeStyles para especificar que no debe agregarse la información de fecha y hora actual al valor DateTime en los campos no especificados.

```
var cultureInfo = new CultureInfo("de-DE");
string dateString = "12 Juni 2008";
var dateTime = DateTime.Parse(dateString, cultureInfo,
                                DateTimeStyles.NoCurrentDateDefault);
Console.WriteLine(dateTime);
// The example displays the following output if the current culture is en-US:
//      6/12/2008 00:00:00
```

```
Dim MyCultureInfo As New CultureInfo("de-DE")
Dim MyString As String = "12 Juni 2008"
Dim MyDateTime As DateTime = DateTime.Parse(MyString, MyCultureInfo,
                           DateTimeStyles.NoCurrentDateDefault)
Console.WriteLine(MyDateTime)
' The example displays the following output if the current culture is en-US:
'       6/12/2008 00:00:00
```

ParseExact

Si se ajusta a uno de los patrones de cadena especificados, el método DateTime.ParseExact convierte una cadena en un objeto DateTime. Cuando se pasa a este método una cadena que no tiene uno de las formas especificadas, se genera una excepción FormatException. Puede definirse uno de los especificadores de formato de fecha y hora estándar o una combinación de los especificadores de formato de fecha y hora personalizados. Con el uso de especificadores de formato personalizados, es posible construir una cadena de reconocimiento personalizada. Para obtener una explicación de los especificadores, consulte los temas sobre cadenas con formato de fecha y hora estándar y cadenas con formato de fecha y hora personalizado.

En el ejemplo siguiente, se pasa al método DateTime.ParseExact un objeto de cadena para que lo analice, seguido de un especificador de formato y luego de un objeto CultureInfo. Este método ParseExact solo puede analizar cadenas que sigan el patrón de fecha larga en la referencia cultural `en-US`.

```
var cultureInfo = new CultureInfo("en-US");
string[] dateStrings = { " Friday, April 10, 2009", "Friday, April 10, 2009" };
foreach (string dateString in dateStrings)
{
    try
    {
        var dateTime = DateTime.ParseExact(dateString, "D", cultureInfo);
        Console.WriteLine(dateTime);
    }
    catch (FormatException)
    {
        Console.WriteLine("Unable to parse '{0}'", dateString);
    }
}
// The example displays the following output:
//       Unable to parse ' Friday, April 10, 2009'
//       4/10/2009 00:00:00
```

```
Dim MyCultureInfo As New CultureInfo("en-US")
Dim MyString() As String = {" Friday, April 10, 2009", "Friday, April 10, 2009"}
For Each dateString As String In MyString
    Try
        Dim MyDateTime As DateTime = DateTime.ParseExact(dateString, "D",
                                                         MyCultureInfo)
        Console.WriteLine(MyDateTime)
    Catch e As FormatException
        Console.WriteLine("Unable to parse '{0}'", dateString)
    End Try
Next
' The example displays the following output:
'       Unable to parse ' Friday, April 10, 2009'
'       4/10/2009 00:00:00
```

Cada sobrecarga de los métodos Parse y ParseExact tiene también un parámetro IFormatProvider que proporciona información específica de la referencia cultural sobre el formato de la cadena. Este objeto IFormatProvider es un objeto CultureInfo que representa una referencia cultural estándar o un objeto DateTimeFormatInfo devuelto por la propiedad CultureInfo.DateTimeFormat. ParseExact usa también una cadena o un argumento de matriz de cadena adicional que define uno o más formatos de fecha y hora personalizados.

Vea también

- Análisis de cadenas
- Aplicar formato a tipos
- Conversión de tipos en .NET
- Cadenas con formato de fecha y hora estándar
- Cadenas con formato de fecha y hora personalizado

Cómo buscar cadenas

29/06/2020 • 7 minutes to read • Edit Online

Puede usar dos estrategias principales para buscar texto en cadenas. Los métodos de la clase String buscan un texto concreto. Las expresiones regulares buscan patrones en el texto.

> **NOTE**
> Los ejemplos de C# de este artículo se ejecutan en el ejecutor de código en línea y área de juegos de Try.NET. Haga clic en el botón **Ejecutar** para ejecutar un ejemplo en una ventana interactiva. Una vez que se ejecuta el código, puede modificar y ejecutar el código modificado si vuelve a hacer clic en **Ejecutar**. El código modificado se ejecuta en la ventana interactiva o, si se produce un error en la compilación, en la ventana interactiva se muestran todos los mensajes de error del compilador de C#.

El tipo string, que es un alias de la clase System.String, proporciona una serie de métodos útiles para buscar el contenido de una cadena. Entre ellos se encuentran Contains, StartsWith, EndsWith, IndexOf y LastIndexOf. La clase System.Text.RegularExpressions.Regex proporciona un vocabulario completo para buscar patrones en el texto. En este artículo, aprenderá estas técnicas y a elegir el mejor método para sus necesidades.

¿Una cadena contiene texto?

Los métodos String.Contains, String.StartsWith y String.EndsWith buscan texto concreto en una cadena. En el ejemplo siguiente, se muestra cada uno de estos métodos y una variación que usa una búsqueda que no distingue mayúsculas de minúsculas:

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Starts with \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult} (ignoring case)");

bool endsWithSearchResult = factMessage.EndsWith(".", System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

En el ejemplo anterior, se muestra un aspecto importante del uso de estos métodos. De manera predeterminada, las búsquedas **distinguen mayúsculas de minúsculas**. Use el valor de enumeración StringComparison.CurrentCultureIgnoreCase para especificar que se trata de una búsqueda que no distingue mayúsculas de minúsculas.

¿Dónde se encuentra el texto buscado en una cadena?

Los métodos IndexOf y LastIndexOf también buscan texto en cadenas. Estos métodos devuelven la ubicación del texto que se busca. Si no se encuentra el texto, devuelven `-1`. En el ejemplo siguiente, se muestra una búsqueda

de la primera y última aparición de la palabra "methods" y muestra el texto que hay en medio.

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\": '{str2}'");
```

Buscar texto concreto mediante expresiones regulares

La clase System.Text.RegularExpressions.Regex se puede usar para buscar cadenas. Estas búsquedas pueden abarcar una complejidad que va desde patrones de texto simples hasta otros complejos.

En el ejemplo de código siguiente, se busca la palabra "the" o "their" en una oración, sin distinción entre mayúsculas y minúsculas. El método estático Regex.IsMatch realiza la búsqueda. Se proporciona la cadena de búsqueda y un patrón de búsqueda. En este caso, un tercer argumento especifica que la búsqueda no distingue mayúsculas de minúsculas. Para obtener más información, vea System.Text.RegularExpressions.RegexOptions.

El patrón de búsqueda describe el texto que se busca. En la tabla siguiente, se describe cada elemento del patrón de búsqueda. (En la tabla siguiente, se usa el valor único `\` que se deben escribir como `\\` en una cadena de C#).

MODELO	SIGNIFICADO
`the`	busca coincidencias con el texto "the"
`(eir)?`	busca coincidencias con 0 o 1 apariciones de "eir"
`\s`	busca coincidencias con un carácter de espacio en blanco

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

TIP

Los métodos `string` suelen ser mejores opciones cuando se busca una cadena exacta. Las expresiones regulares son más adecuadas cuando se busca algún patrón en una cadena de origen.

¿Una cadena sigue un patrón?

El código siguiente usa expresiones regulares para validar el formato de cada cadena de una matriz. La validación requiere que cada cadena tenga la forma de un número de teléfono en el que tres grupos de dígitos se separan por guiones. Los dos primeros grupos contienen tres dígitos y el tercero, cuatro. El patrón de búsqueda usa la expresión regular `^\\d{3}-\\d{3}-\\d{4}$`. Para obtener más información, consulte Lenguaje de expresiones regulares: Referencia rápida.

MODELO	SIGNIFICADO
`^`	busca coincidencias con el principio de la cadena
`\d{3}`	busca coincidencias con exactamente 3 caracteres de dígitos
`-`	busca coincidencias con el carácter "-"
`\d{3}`	busca coincidencias con exactamente 3 caracteres de dígitos
`-`	busca coincidencias con el carácter "-"
`\d{4}`	busca coincidencias con exactamente 4 caracteres de dígitos
`$`	busca coincidencias con el final de la cadena

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";

foreach (string s in numbers)
{
    Console.Write($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

Este patrón de búsqueda sencillo coincide con muchas cadenas válidas. Las expresiones regulares son mejores para buscar o validar con respecto a un patrón, en lugar de una cadena de texto sencilla.

Vea también

- Guía de programación de C#
- Cadenas
- LINQ y cadenas
- System.Text.RegularExpressions.Regex
- Expresiones regulares de .NET Framework
- Lenguaje de expresiones regulares: referencia rápida
- Procedimientos recomendados para el uso de cadenas en .NET

Procedimiento para modificar el contenido de cadenas en C#

29/06/2020 • 9 minutes to read • Edit Online

En este artículo se muestran varias técnicas para producir una `string` modificando una `string` existente. Todas las técnicas mostradas devuelven el resultado de las modificaciones como un objeto `string` nuevo. Para indicarlo de forma clara, en todos los ejemplos se almacena el resultado en una variable nueva. Entonces, podrá examinar tanto la `string` original como la `string` resultante de la modificación al ejecutar cada ejemplo.

> **NOTE**
>
> Los ejemplos de C# de este artículo se ejecutan en el ejecutor de código en línea y área de juegos de Try.NET. Haga clic en el botón **Ejecutar** para ejecutar un ejemplo en una ventana interactiva. Una vez que se ejecuta el código, puede modificar y ejecutar el código modificado si vuelve a hacer clic en **Ejecutar**. El código modificado se ejecuta en la ventana interactiva o, si se produce un error en la compilación, en la ventana interactiva se muestran todos los mensajes de error del compilador de C#.

En este artículo se muestran varias técnicas. Puede reemplazar el texto existente. Puede buscar patrones y reemplazar el texto coincidente por otro texto. Puede tratar una cadena con una secuencia de caracteres. También puede usar métodos de conveniencia para eliminar espacios en blanco. Elija la técnica con mayor coincidencia con el escenario.

Reemplazo de texto

Con el código siguiente se crea una cadena mediante el reemplazo de texto con un sustituto.

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

En el código anterior se muestra esta propiedad *inmutable* de las cadenas. En el ejemplo anterior puede ver que la cadena original, `source`, no se ha modificado. Con el método String.Replace se crea una `string` que contiene las modificaciones.

Con el método Replace se pueden reemplazar cadenas o caracteres únicos. En ambos casos, se reemplazan todas las instancias del texto buscado. En el siguiente ejemplo se reemplazan todos los caracteres " " por "_":

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

La cadena de origen se mantiene y se devuelve una cadena nueva con los reemplazos.

Recorte de espacios en blanco

Puede usar los métodos String.Trim, String.TrimStart, y String.TrimEnd para quitar los espacios en blanco al inicio y al final. En el código siguiente se muestra un ejemplo de cada caso. La cadena de origen no cambia; con estos métodos se devuelve una cadena nueva con el contenido modificado.

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.      ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

Eliminación de texto

Puede quitar texto de una cadena con el método String.Remove. Este método quita un número de caracteres que comienzan con un índice específico. En el siguiente ejemplo se muestra cómo usar String.IndexOf seguido por Remove para quitar texto de una cadena:

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

Reemplazo de patrones de coincidencia

Puede usar expresiones regulares para reemplazar texto que coincida con patrones por texto nuevo, posiblemente definido por un patrón. En el ejemplo siguiente se usa la clase System.Text.RegularExpressions.Regex para encontrar un patrón en una cadena de origen y reemplazarlo con un uso de mayúsculas y minúsculas adecuado. Con el método Regex.Replace(String, String, MatchEvaluator, RegexOptions) se usa una función que proporciona la lógica del reemplazo de uno de los argumentos. En este ejemplo, la función `LocalReplaceMatchCase` es una **función local** declarada dentro del método de ejemplo. `LocalReplaceMatchCase` usa la clase System.Text.StringBuilder para crear la cadena de reemplazo con un uso de mayúsculas y minúsculas adecuado.

Las expresiones regulares son más útiles al buscar y reemplazar texto que sigue un patrón, en vez de texto que ya conoce. Para obtener más información, vea Procedimiento para buscar cadenas. Con el patrón de búsqueda "the\s" se busca la palabra "the" seguida de un carácter de espacio en blanco. Con esa parte del patrón se asegura de que no se busca "there" en la cadena de origen. Para obtener más información sobre los elementos de lenguaje de expresiones regulares, vea Lenguaje de expresiones regulares - Referencia rápida.

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s", LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

Con el método StringBuilder.ToString se devuelve una cadena inmutable con el contenido del objeto StringBuilder.

Modificación de caracteres individuales

Puede producir un matriz de caracteres a partir de una cadena, modificar el contenido de la matriz y crear después una cadena a partir del contenido modificado de la matriz.

En el siguiente ejemplo se muestra cómo reemplazar un conjunto de caracteres en una cadena. En primer lugar, se usa el método String.ToCharArray() para crear una matriz de caracteres. Se usa el método IndexOf para encontrar el índice de inicio de la palabra "fox". Los siguientes tres caracteres se reemplazan por otra palabra. Por último, se construye una cadena nueva a partir de la matriz de carácter actualizada.

```
string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

Creación mediante programación del contenido de la cadena

Dado que las cadenas son inmutables, en los ejemplos anteriores se crean cadenas temporales o matrices de caracteres. En escenarios de alto rendimiento, puede ser conveniente evitar estas asignaciones de montón. .NET Core proporciona un método String.Create que permite rellenar mediante programación el contenido de los caracteres de una cadena a través de una devolución de llamada, a la vez que evita las asignaciones de cadenas temporales intermedias.

```
// constructing a string from a char array, prefix it with some additional characters
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
int length = chars.Length + 2;
string result = string.Create (length, chars, (Span<char> strContent, char[] charArray) = >
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

Puede modificar una cadena en un bloque fijo con código no seguro, pero es **totalmente** desaconsejable modificar el contenido de la cadena una vez que se ha creado. Si lo hace, puede haber problemas imprevisibles. Por ejemplo, si alguien se conecta a una cadena que tiene el mismo contenido que el suyo, obtendrá una copia y no esperará que usted modifique la cadena.

Vea también

- Expresiones regulares de .NET Framework
- Lenguaje de expresiones regulares: referencia rápida

Cómo comparar cadenas en C#

29/06/2020 • 18 minutes to read • Edit Online

Se comparan cadenas para responder a una de estas dos preguntas: "¿Son iguales estas dos cadenas?" o "¿En qué orden deben colocarse estas cadenas al ordenarlas?"

Esas dos preguntas se complican por factores que influyen en las comparaciones de cadenas:

- Puede elegir una comparación ordinal o lingüística.
- Puede elegir si distingue entre mayúsculas y minúsculas.
- Puede elegir comparaciones específicas de referencia cultural.
- Las comparaciones lingüísticas dependen de la plataforma y la referencia cultural.

> **NOTE**
>
> Los ejemplos de C# de este artículo se ejecutan en el ejecutor de código en línea y área de juegos de Try.NET. Haga clic en el botón **Ejecutar** para ejecutar un ejemplo en una ventana interactiva. Una vez que se ejecuta el código, puede modificar y ejecutar el código modificado si vuelve a hacer clic en **Ejecutar**. El código modificado se ejecuta en la ventana interactiva o, si se produce un error en la compilación, en la ventana interactiva se muestran todos los mensajes de error del compilador de C#.

Cuando se comparan cadenas, se define un orden entre ellas. Las comparaciones se usan para ordenar una secuencia de cadenas. Una vez que la secuencia está en un orden conocido, es más fácil hacer búsquedas, tanto para el software como para las personas. Otras comparaciones pueden comprobar si las cadenas son iguales. Estas comprobaciones de similitud son parecidas a la igualdad, pero pueden omitirse algunas diferencias, como las diferencias entre mayúsculas y minúsculas.

Comparaciones de ordinales predeterminadas

De forma predeterminada, las operaciones más comunes:

- String.CompareTo
- String.Equals
- String.Equality y String.Inequality; es decir, los operadores de igualdad `==` y `!=`, respectivamente,

realizan una comparación ordinal con distinción entre mayúsculas y minúsculas y, si es necesario, usan la referencia cultural actual. En el siguiente ejemplo se muestra que:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root == root2 ? "equal" : "not equal")}");
```

La comparación de ordinales predeterminada no tiene en cuenta reglas lingüísticas cuando se comparan cadenas. Compara el valor binario de cada objeto Char en dos cadenas. Como resultado, la comparación de ordinales

predeterminada también distingue mayúsculas de minúsculas.

Tenga en cuenta que la prueba de igualdad con String.Equals y los operadores `==` y `!=` es diferente de la comparación de cadenas que usa los métodos String.CompareTo y Compare(String, String). Mientras que las pruebas de igualdad realizan una comparación ordinal que distingue mayúsculas de minúsculas, los métodos de comparación realizan una comparación que distingue mayúsculas de minúsculas y entre referencias culturales usando la referencia cultural actual. Puesto que los métodos de comparación predeterminados suelen realizan diferentes tipos de comparaciones, le recomendamos que especifique claramente la intención de su código llamando a una sobrecarga que especifique explícitamente el tipo de comparación que se realizará.

Comparaciones de ordinales sin distinción entre mayúsculas y minúsculas

El método String.Equals(String, StringComparison) le permite especificar un valor StringComparison de StringComparison.OrdinalIgnoreCase para una comparación ordinal sin distinción entre mayúsculas y minúsculas. También hay un método String.Compare(String, String, StringComparison) estático que realiza una comparación ordinal que distingue mayúsculas de minúsculas. Para usarlo, debe especificar un valor de StringComparison.OrdinalIgnoreCase para el argumento StringComparison. Se muestran en este código:

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2, StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType: StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are {(areEqual ? "equal." : "not
equal.")}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
```

Al realizar una comparación ordinal que distingue mayúsculas de minúsculas, estos métodos usarán las convenciones de mayúsculas y minúsculas de la referencia cultural invariable.

Comparaciones lingüísticas

También se pueden ordenar cadenas mediante reglas lingüísticas para la referencia cultural actual. Esto se conoce a veces como "criterio de ordenación por palabras". Cuando se realiza una comparación lingüística, algunos caracteres Unicode no alfanuméricos pueden tener asignados pesos especiales. Por ejemplo, el guion ("-") podría tener asignado un peso muy pequeño, por lo que las cadenas "coop" y "co-op" aparecerían una junto a la otra en una ordenación. Además, algunos caracteres Unicode pueden ser equivalentes a una secuencia de instancias de Char. En este ejemplo se usa una frase en alemán que significa "Bailan en la calle", en alemán, con "ss" (U+0073 U+0073) en una cadena y "ß" (U+00DF) en otra. Lingüísticamente (en Windows), "ss" es igual que el carácter "ß" en alemán en las referencias culturales "en-US" y "de-DE".

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second, StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")} equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two, StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}
```

Este ejemplo demuestra la naturaleza dependiente del sistema operativo de las comparaciones lingüísticas. El host de la ventana interactiva es un host Linux. Las comparaciones lingüísticas y ordinales producen el mismo resultado. Si se ejecutara este mismo ejemplo en un host de Windows, vería este resultado:

```
<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison
```

En Windows, el criterio de ordenación de "cop", "coop" y "co-op" cambia al cambiar de una comparación lingüística a una comparación ordinal. Las dos frases en alemán también se comparan de manera diferente mediante tipos de comparación diferentes.

Comparaciones con referencias culturales específicas

En esta muestra se almacenan los objetos CultureInfo de las referencias culturales en-US y de-DE. Las comparaciones se realizan con el objeto CultureInfo para garantizar una comparación específica de la referencia cultural.

La referencia cultural usada afecta a las comparaciones lingüísticas. En este ejemplo se muestra el resultado de comparar las dos frases en alemán usando la referencia cultural "en-US" y la referencia cultural "de-DE":

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de, System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo culture)
{
    int compareLinguistic = String.Compare(one, two, en, System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order using ordinal comparison");
}
```

Las comparaciones dependientes de la referencia cultural se usan normalmente para comparar y ordenar cadenas escritas por usuarios con otras cadenas escritas por usuarios. Los caracteres y las convenciones de ordenación de estas cadenas pueden variar según la configuración regional del equipo del usuario. Incluso las cadenas que contienen caracteres idénticos podrían ordenarse de forma diferente en función de la referencia cultural del subproceso actual. Además, pruebe este código de ejemplo localmente en una máquina con Windows y obtendrá estos resultados:

```
<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison
```

Las comparaciones lingüísticas dependen de la referencia cultural actual y del sistema operativo. Debe tenerlo en cuenta cuando trabaje con comparaciones de cadenas.

Ordenación lingüística y búsqueda de cadenas en matrices

En estos ejemplos se muestra cómo ordenar y buscar cadenas en una matriz mediante una comparación lingüística que depende de la referencia cultural actual. Use los métodos Array estáticos que toman un parámetro System.StringComparer.

En este ejemplo se muestra cómo ordenar una matriz de cadenas con la referencia cultural actual:

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Una vez que se ordena la matriz, puede buscar entradas mediante una búsqueda binaria. Una búsqueda binaria empieza en medio de la colección para determinar qué mitad de la colección debe contener la cadena buscada. Cada comparación posterior divide la parte restante de la colección por la mitad. La matriz se ordena con el elemento StringComparer.CurrentCulture. La función local `ShowWhere` muestra información sobre dónde se encuentra la cadena. Si no se encuentra la cadena, el valor devuelto indica dónde estaría si se encontrara.

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString, StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Ordenación de ordinales y búsqueda en colecciones

Este código usa la clase de colección System.Collections.Generic.List<T> para almacenar cadenas. Las cadenas se ordenan mediante el método List<T>.Sort. Este método necesita un delegado que compare y ordene dos cadenas. El método String.CompareTo proporciona esa función de comparación. Ejecute el ejemplo y observe el orden. Esta operación de ordenación usa una ordenación ordinal con distinción entre mayúsculas y minúsculas. Tendría que usar los métodos estáticos String.Compare para especificar reglas de comparación distintas.

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Una vez realizada la ordenación, se pueden hacer búsquedas en la lista de cadenas mediante una búsqueda binaria. En este ejemplo se muestra cómo buscar la lista ordenada usando la misma función de comparación. La función local `ShowWhere` muestra dónde está o debería estar el texto buscado:

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")} {searchString}");

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Asegúrese siempre de usar el mismo tipo de comparación para la ordenación y la búsqueda. Si se usan distintos tipos de comparación para la ordenación y la búsqueda se producen resultados inesperados.

Las clases de colección como System.Collections.Hashtable, System.Collections.Generic.Dictionary<TKey,TValue> y System.Collections.Generic.List<T> tienen constructores que toman un parámetro System.StringComparer cuando el tipo de los elementos o claves es `string`. En general, debe usar estos constructores siempre que sea posible y especificar StringComparer.Ordinal u StringComparer.OrdinalIgnoreCase.

Igualdad de referencia e internamiento de cadena

Ninguno de los ejemplos ha usado ReferenceEquals. Este método determina si dos cadenas son el mismo objeto. Esto puede conducir a resultados incoherentes en comparaciones de cadenas. En este ejemplo se muestra la característica de *internamiento de cadenas* de C#. Cuando un programa declara dos o más variables de cadena idénticas, el compilador lo almacena todo en la misma ubicación. Mediante una llamada al método ReferenceEquals, puede ver que las dos cadenas realmente hacen referencia al mismo objeto en memoria. Use el método String.Copy para evitar el internamiento. Después de hacer la copia, las dos cadenas tienen diferentes ubicaciones de almacenamiento, aunque tengan el mismo valor. Ejecute este ejemplo para mostrar que las cadenas `a` y `b` son *internadas*, lo que significa que comparten el mismo almacenamiento. Las cadenas `a` y `c` no lo son.

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

NOTE

Cuando se prueba la igualdad de cadenas, debe usar los métodos que especifican explícitamente el tipo de comparación que va a realizar. El código se vuelve mucho más legible y fácil de mantener. Use sobrecargas de los métodos de las clases System.String y System.Array que toman un parámetro de enumeración StringComparison. Especifique qué tipo de comparación se va a realizar. Evite usar los operadores `==` y `!=` cuando pruebe la igualdad. Los métodos de instancia String.CompareTo siempre realizan una comparación ordinal con distinción entre mayúsculas y minúsculas. Son adecuados principalmente para ordenar alfabéticamente las cadenas.

Puede internalizar una cadena o recuperar una referencia a una cadena internalizada existente llamando al método String.Intern. Para determinar si una cadena se aplica el método Intern, llame al método String.IsInterned.

Vea también

- System.Globalization.CultureInfo
- System.StringComparer
- Cadenas
- Comparación de cadenas
- Globalizar y localizar aplicaciones

Procedimiento para convertir de forma segura mediante la coincidencia de patrones y los operadores is y as

29/06/2020 • 7 minutes to read • Edit Online

Dado que los objetos son polimórficos, es posible que una variable de un tipo de clase base contenga un tipo derivado. Para acceder a los miembros de instancia del tipo derivado, es necesario volver a convertir el valor en el tipo derivado. Pero una conversión conlleva el riesgo de producir una InvalidCastException. C# proporciona instrucciones de coincidencia de patrones que realizan una conversión condicionalmente, solo si se va a realizar correctamente. C# además proporciona los operadores is y as para probar si un valor es de un tipo determinado.

En el ejemplo siguiente se muestra cómo usar la instrucción `is` de coincidencia de patrones.

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        var g = new Giraffe();
        var a = new Animal();
        FeedMammals(g);
        FeedMammals(a);
        // Output:
        // Eating.
        // Animal is not a Mammal

        SuperNova sn = new SuperNova();
        TestForMammals(g);
        TestForMammals(sn);
        // Output:
        // I am an animal.
        // SuperNova is not a Mammal
    }

    static void FeedMammals(Animal a)
    {
        if (a is Mammal m)
        {
            m.Eat();
        }
        else
        {
            // variable 'm' is not in scope here, and can't be used.
            Console.WriteLine($"{a.GetType().Name} is not a Mammal");
        }
    }

    static void TestForMammals(object o)
    {
        // You also can use the as operator and test for null
        // before referencing the variable.
        var m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }
}
```

El ejemplo anterior muestra una serie de características de sintaxis de coincidencia de patrones. La instrucción `if (a is Mammal m)` combina la prueba con una asignación de inicialización. La asignación solo se produce cuando la prueba se realiza correctamente. La variable `m` solo está en ámbito en la instrucción `if` insertada donde se ha

asignado. No se puede acceder a `m` más adelante en el mismo método. En el ejemplo anterior también se muestra cómo usar el operador `as` para convertir un objeto en un tipo especificado.

También puede usar la misma sintaxis para probar si un tipo de valor que admite valores NULL tiene un valor, como se muestra en el ejemplo siguiente:

```
class Program
{
    static void Main(string[] args)
    {
        int i = 5;
        PatternMatchingNullable(i);

        int? j = null;
        PatternMatchingNullable(j);

        double d = 9.78654;
        PatternMatchingNullable(d);

        PatternMatchingSwitch(i);
        PatternMatchingSwitch(j);
        PatternMatchingSwitch(d);
    }

    static void PatternMatchingNullable(System.ValueType val)
    {
        if (val is int j) // Nullable types are not allowed in patterns
        {
            Console.WriteLine(j);
        }
        else if (val is null) // If val is a nullable type with no value, this expression is true
        {
            Console.WriteLine("val is a nullable type with the null value");
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }

    static void PatternMatchingSwitch(System.ValueType val)
    {
        switch (val)
        {
            case int number:
                Console.WriteLine(number);
                break;
            case long number:
                Console.WriteLine(number);
                break;
            case decimal number:
                Console.WriteLine(number);
                break;
            case float number:
                Console.WriteLine(number);
                break;
            case double number:
                Console.WriteLine(number);
                break;
            case null:
                Console.WriteLine("val is a nullable type with the null value");
                break;
            default:
                Console.WriteLine("Could not convert " + val.ToString());
                break;
        }
    }
}
```

El ejemplo anterior muestra otras características de coincidencia de patrones para usar con conversiones. Puede probar una variable para el patrón null si busca específicamente el valor `null`. Cuando el valor de runtime de la variable es `null`, una instrucción `is` que busca un tipo siempre devuelve `false`. La instrucción `is` de

coincidencia de patrones no permite un tipo de valor que acepta valores NULL, como `int?` o `Nullable<int>`, pero puede probar con cualquier otro tipo de valor. Los patrones `is` del ejemplo anterior no se limitan a los tipos de valor que admiten un valor NULL. También puede usar esos patrones para probar si una variable de un tipo de referencia tiene un valor o es `null`.

En el ejemplo anterior también se muestra cómo usar el patrón de tipo en una instrucción `switch` donde la variable puede ser uno de muchos tipos diferentes.

Si quiere probar si una variable es de un tipo determinado, pero no asignarla a una nueva variable, puede usar los operadores `is` y `as` para los tipos de referencia y los tipos que aceptan valores NULL. El código siguiente muestra cómo usar las instrucciones `is` y `as` que formaban parte del lenguaje C# antes de la incorporación de la coincidencia de patrones para probar si una variable es de un tipo determinado:

```
class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

class Program
{
    static void Main(string[] args)
    {
        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        UseAsOperator(sn);

        // Use the as operator with a value type.
        // Note the implicit conversion to int? in
        // the method body.
        int i = 5;
        UseAsWithNullable(i);

        double d = 9.78654;
        UseAsWithNullable(d);
    }

    static void UseIsOperator(Animal a)
    {
        if (a is Mammal)
        {
            Mammal m = (Mammal)a;
            m.Eat();
        }
    }

    static void UsePatternMatchingIs(Animal a)
    {
```

```
        if (a is Mammal m)
        {
            m.Eat();
        }
    }

    static void UseAsOperator(object o)
    {
        Mammal m = o as Mammal;
        if (m != null)
        {
            Console.WriteLine(m.ToString());
        }
        else
        {
            Console.WriteLine($"{o.GetType().Name} is not a Mammal");
        }
    }

    static void UseAsWithNullable(System.ValueType val)
    {
        int? j = val as int?;
        if (j != null)
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("Could not convert " + val.ToString());
        }
    }
}
```

Como puede ver al comparar este código con el código de coincidencia de patrones, la sintaxis de coincidencia de patrones proporciona características más sólidas mediante la combinación de la prueba y la asignación en una sola instrucción. Use la sintaxis de coincidencia de patrones siempre que sea posible.

SDK de .NET Compiler Platform

18/03/2020 • 12 minutes to read • Edit Online

Los compiladores crean un modelo detallado de código de aplicación a medida que validan la sintaxis y semántica de ese código. Utilizan este modelo para generar la salida ejecutable desde el código fuente. El SDK de .NET Compiler Platform proporciona acceso a este modelo. Cada vez confiamos más en las características del entorno de desarrollo integrado (IDE), como IntelliSense, la refactorización, el cambio de nombre inteligente, "Buscar todas las referencias" e "Ir a definición" para aumentar la productividad. Nos basamos en herramientas de análisis de código para mejorar la calidad de nuestro código y en generadores de código para ayudar en la construcción de la aplicación. A medida que la inteligencia de estas herramientas aumenta, cada vez necesitan más acceso del modelo que solo los compiladores crean cuando procesan el código de la aplicación. Se trata de la misión principal de las API de Roslyn: abrir las cajas negras y permitir a las herramientas y a los usuarios finales compartir en la gran cantidad de información que los compiladores tienen sobre nuestro código. En lugar de traductores opacos de entrada de código fuente y salida de código objeto, a través de Roslyn, los compiladores se convierten en plataformas: las API que puede usar para tareas relacionadas con el código en las herramientas y aplicaciones.

Conceptos del SDK de .NET Compiler Platform

El SDK de .NET Compiler Platform reduce notablemente la barrera de entrada para crear herramientas y aplicaciones centradas en código. Crea numerosas oportunidades para la innovación en áreas como la metaprogramación, la generación y la transformación de código, el uso interactivo de los lenguajes C# y Visual Basic, y la inserción de C# y Visual Basic en lenguajes específicos del dominio.

El SDK de .NET Compiler Platform le permite crear ***analizadores*** y ***correcciones de código*** que buscan y corrigen errores de codificación. Los ***analizadores*** comprenden la sintaxis y la estructura del código y detectan las prácticas que se deben corregir. Las ***correcciones de código*** sugieren una o varias correcciones para tratar los errores de codificación que los analizadores encuentran. Por lo general, un analizador y las correcciones de código asociadas se empaquetan conjuntamente en un solo proyecto.

Los analizadores y las correcciones de código usan el análisis estático para comprender el código. No ejecutan el código o proporcionan otras ventajas de pruebas. Sin embargo, pueden señalar prácticas que suelen dar lugar a errores, código que no se puede mantener o validación de guías estándar.

El SDK de .NET Compiler Platform proporciona un único conjunto de API que le permite examinar y entender un código base de C# o Visual Basic. Dado que puede usar este código base único, puede escribir analizadores y correcciones de código más fácilmente aprovechando las API de análisis sintáctico y semántico que proporciona el SDK de .NET Compiler Platform. Una vez liberado de la ardua tarea de replicar el análisis realizado por el compilador, puede concentrarse en la tarea más específica de encontrar y corregir errores de codificación comunes para el proyecto o la biblioteca.

Una ventaja menor es que los analizadores y las correcciones de código son más pequeños y usan muchos menos memoria cuando se cargan en Visual Studio que si escribiera su propio código base para entender el código de un proyecto. Aprovechando las mismas clases que usa el compilador y Visual Studio, puede crear sus propias herramientas de análisis estático. Esto significa que el equipo puede usar los analizadores y las correcciones de código sin un impacto perceptible en el rendimiento del IDE.

Hay tres escenarios principales para escribir analizadores y correcciones de código:

1. *Aplicar estándares de codificación de equipo*
2. *Proporcionar instrucciones con paquetes de biblioteca*
3. *Suministro de una guía general*

Aplicación de estándares de codificación de equipo

Muchos equipos tienen estándares de codificación que se aplican a través de revisiones del código con otros miembros del equipo. Los analizadores y las correcciones de código pueden hacer este proceso mucho más eficiente. Las revisiones de código se producen cuando un desarrollador comparte su trabajo con otros usuarios en el equipo. Dicho desarrollador habrá invertido todo el tiempo necesario para completar una nueva característica antes de obtener los comentarios. Pueden pasar semanas mientras que refuerza los hábitos que no coinciden con las prácticas del equipo.

Los analizadores se ejecutan a medida que un desarrollador escribe código. Obtiene comentarios al instante que animan a seguir la guía de inmediato. Crea hábitos para escribir código compatible tan pronto como comienza la creación de prototipos. Cuando la característica está lista para que las personas la revisen, se habrá aplicado toda la guía estándar.

Los equipos pueden crear analizadores y correcciones de código que busquen las prácticas más comunes que infringen las prácticas de codificación del equipo. Se pueden instalar en el equipo del desarrollador para aplicar los estándares.

Suministro de instrucciones con paquetes de biblioteca

Hay una gran cantidad de bibliotecas disponibles para los desarrolladores de .NET en NuGet. Algunos de ellas proceden de Microsoft, algunas de compañías de terceros y otras de voluntarios y miembros de la comunidad. Estas bibliotecas consiguen más adopción y revisiones superiores cuando los desarrolladores pueden tener éxito con ellas.

Además de proporcionar documentación, puede proporcionar analizadores y correcciones de código que busquen y corrijan los usos incorrectos habituales de la biblioteca. Estas correcciones inmediatas ayudarán a los desarrolladores a tener éxito más rápidamente.

Puede empaquetar los analizadores y las correcciones de código con la biblioteca en NuGet. En ese escenario, todos los desarrolladores que instalan el paquete de NuGet también instalarán el paquete de analizador. Todos los programadores que utilicen la biblioteca obtendrán instrucciones de su equipo al instante en forma de comentarios inmediatos sobre errores y correcciones sugeridas.

Suministro de una guía general

La comunidad de desarrolladores de .NET ha detectado patrones de experiencia que funcionan bien y patrones que es mejor evitar. Varios miembros de la comunidad han creado analizadores que aplican esos patrones recomendados. A medida que aprendemos más, siempre hay espacio para nuevas ideas.

Estos analizadores se puedan cargar en Visual Studio Marketplace y los desarrolladores los pueden descargar mediante Visual Studio. Los recién llegados al lenguaje y a la plataforma aprenden rápidamente las prácticas aceptadas y consiguen ser productivos antes en su recorrido por .NET. A medida que su uso se amplía, la comunidad adopta estas prácticas.

Pasos siguientes

El SDK de .NET Compiler Platform incluye los modelos de objetos de idioma más recientes para generación de código, análisis y refactorización. Esta sección proporciona información general conceptual del SDK de .NET Compiler Platform. Encontrará más detalles en las secciones de guías de inicio rápido, ejemplos y tutoriales.

Puede obtener más información sobre los conceptos del SDK de .NET Compiler Platform en estos cinco temas:

- Exploración de código con el visualizador de sintaxis
- Entender el modelo de API de compilador

- Trabajar con sintaxis
- Trabajar con semántica
- Trabajar con un área de trabajo

Para empezar, debe instalar el **SDK de .NET Compiler Platform**:

Instrucciones de instalación: Instalador de Visual Studio

Hay dos maneras distintas de buscar el **SDK de .NET Compiler Platform** en el **Instalador de Visual Studio**:

Instalación con el Instalador de Visual Studio: visualización de cargas de trabajo

El SDK de .NET Compiler Platform no se selecciona automáticamente como parte de la carga de trabajo de desarrollo de extensiones de Visual Studio. Se debe seleccionar como un componente opcional.

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Active la carga de trabajo **Desarrollo de extensiones de Visual Studio**.
4. Abra el nodo **Desarrollo de extensiones de Visual Studio** en el árbol de resumen.
5. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en última posición bajo los componentes opcionales.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Abra el nodo **Componentes individuales** en el árbol de resumen.
2. Active la casilla **Editor de DGML**.

Instalación con el Instalador de Visual Studio: pestaña Componentes individuales

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Haga clic en la pestaña **Componentes individuales**.
4. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en la parte superior bajo la sección **Compiladores, herramientas de compilación y tiempos de ejecución**.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Active la casilla **Editor de DGML**. La encontrará en la sección **Herramientas de código**.

Descripción del modelo del SDK de .NET Compiler Platform

12/05/2020 • 7 minutes to read • Edit Online

Los compiladores procesan el código que se escribe conforme a reglas estructuradas que a menudo se diferencian de la forma en que los seres humanos lo leen y lo entienden. Una comprensión básica del modelo usado por los compiladores resulta fundamental para entender las API que se usan al compilar herramientas basadas en Roslyn.

Áreas funcionales de canalización de compilador

El SDK de .NET Compiler Platform expone el análisis de código de los compiladores de C# y Visual Basic al proporcionar una capa de API que refleja una canalización de compilador tradicional.

Cada fase de esta canalización es un componente independiente. En primer lugar, la fase de análisis acorta y analiza el texto de origen en la sintaxis que sigue la gramática del lenguaje. En segundo lugar, la fase de declaración analiza los metadatos importados y de origen para formar símbolos con nombre. Luego, la fase de enlace combina los identificadores del código con los símbolos. Por último, en la fase de emisión se emite un ensamblado con toda la información generada por el compilador.

En correspondencia a cada una de esas fases, el SDK de .NET Compiler Platform expone un modelo de objetos que permite el acceso a la información en esa fase. La fase de análisis expone un árbol de sintaxis, la fase de declaración expone una tabla de símbolos jerárquica, la fase de enlace expone el resultado del análisis semántico del compilador y la fase de emisión es una API que genera códigos de bytes de IL.

Cada compilador combina estos componentes como un único todo.

Estas API son las mismas que usa Visual Studio. Por ejemplo, las características de formato y esquema del código usan los árboles de sintaxis, las características de navegación y el Examinador de objetos usan la tabla de símbolos, las refactorizaciones e Ir a definición usan el modelo semántico y Editar y continuar usa todos ellos, incluida la API de emisión.

Capas de API

El SDK de .NET Compiler está formado por dos capas principales de API: API de compilador y API de áreas de trabajo.

API de compilador

La capa de compilador contiene los modelos de objetos que corresponden con la información expuesta en cada fase de la canalización de compilador, sintáctica y semántica. La capa de compilador también contiene una instantánea inmutable de una sola invocación de un compilador, incluidas las referencias de ensamblado, las opciones del compilador y los archivos de código fuente. Hay dos API distintas que representan el lenguaje C# y el lenguaje Visual Basic. Estas dos API son similares en forma, aunque están adaptadas para lograr una alta fidelidad con cada lenguaje. Esta capa no tiene dependencias en componentes de Visual Studio.

API de diagnóstico

Como parte de su análisis, el compilador puede generar un conjunto de diagnósticos que abarcan desde errores de sintaxis, semántica y asignación definitiva hasta distintas advertencias y diagnósticos informativos. La capa de la API de compilador expone diagnósticos a través de una API extensible que permite incluir analizadores definidos por el usuario en el proceso de compilación. Permite generar diagnósticos definidos por el usuario, como los de

herramientas como StyleCop o FxCop, junto con diagnósticos definidos por el compilador. La generación de diagnósticos de este modo tiene la ventaja de integrarse de forma natural con herramientas como MSBuild y Visual Studio que dependen de diagnósticos de experiencias como detener una compilación según una directiva y mostrar subrayados ondulados en vivo en el editor y sugerir correcciones de código.

API de scripting

Las API de hospedaje y scripting forman parte de la capa de compilador. Puede usarlas para ejecutar fragmentos de código y acumular un contexto de ejecución en tiempo de ejecución. El REPL (bucle de lectura, evaluación e impresión) interactivo de C# usa estas API. El REPL permite usar C# como lenguaje de scripting, al ejecutar el código de forma interactiva mientras se escribe.

API de áreas de trabajo

La capa de áreas de trabajo contiene la API de área de trabajo, que es el punto de partida para realizar análisis de código y refactorización de soluciones completas. Ayuda a organizar toda la información sobre los proyectos de una solución en un modelo de objetos único, lo que ofrece acceso directo a los modelos de objetos de capa de compilador sin necesidad de analizar archivos, configurar opciones ni administrar dependencias entre proyectos.

Además, la capa de áreas de trabajo expone un conjunto de API que se usa al implementar las herramientas de análisis de código y refactorización que funcionan en un entorno de host como el IDE de Visual Studio. Los ejemplos incluyen las API Buscar todas las referencias, Formato y Generación de código.

Esta capa no tiene dependencias en componentes de Visual Studio.

Trabajar con sintaxis

29/06/2020 • 17 minutes to read • Edit Online

El *árbol de sintaxis* es una estructura de datos fundamental expuesta por las API del compilador. Estos árboles representan la estructura léxica y sintáctica del código fuente. Tienen dos importantes finalidades:

- Permitir herramientas, como un IDE, complementos, herramientas de análisis de código y refactorizaciones, y ver y procesar la estructura sintáctica del código fuente del proyecto de un usuario.
- Habilitar herramientas, como refactorizaciones y un IDE, para crear, modificar y reorganizar el código fuente de forma natural sin tener que usar ediciones de texto directas. Al crear y manipular árboles, las herramientas pueden crear y reorganizar fácilmente el código fuente.

Árboles de sintaxis

Los árboles de sintaxis son la estructura principal usada para la compilación, el análisis de código, los enlaces, la refactorización, las características de IDE y la generación de código. Ninguna parte del código fuente se entiende sin que primero se haya identificado y clasificado en alguno de los elementos de lenguaje estructural conocidos.

Los árboles de sintaxis tienen tres atributos clave. El primer atributo es que los árboles de sintaxis contienen toda la información de origen con plena fidelidad. Esto significa que el árbol de sintaxis contiene cada fragmento de información del texto de origen, cada construcción gramatical, cada token léxico y todo lo demás, incluidos los espacios en blanco, los comentarios y las directivas de preprocesador. Por ejemplo, cada literal mencionado en el origen se representa exactamente como se ha escrito. Los árboles de sintaxis también capturan los errores del código fuente cuando el programa está incompleto o tiene un formato incorrecto mediante la representación de los tokens omitidos o que faltan.

El segundo atributo de los árboles de sintaxis es que pueden generar el texto exacto desde el que se analizaron. Es posible obtener la representación de texto del subárbol cuya raíz está en ese nodo desde cualquier nodo de sintaxis. Esta posibilidad significa que los árboles de sintaxis se pueden usar como una manera de crear y editar texto de origen. Al crear un árbol, ha creado de manera implícita el texto equivalente, mientras que al editar un árbol de sintaxis o crear uno a partir de los cambios en uno existente, ha editado realmente el texto.

El tercer atributo de los árboles de sintaxis es que son inmutables y seguros para subprocesos. Una vez obtenido un árbol, es una instantánea del estado actual del código y nunca cambia. Esto permite que varios usuarios interactúen con el mismo árbol de sintaxis a la vez en distintos subprocesos sin que se produzca ningún bloqueo ni duplicación. Dado que los árboles son inmutables y no permiten ninguna modificación directa, los métodos de fábrica ayudan a crear y modificar los árboles de sintaxis mediante la creación de instantáneas adicionales del árbol. Los árboles son eficaces en su forma de volver a usar nodos subyacentes, así que es posible volver a crear una nueva versión rápidamente y con poca memoria adicional.

Un árbol de sintaxis es literalmente una estructura de datos de árbol donde los elementos estructurales no terminales son primarios con respecto a otros elementos. Cada árbol de sintaxis se compone de nodos, tokens y curiosidades.

Nodos de sintaxis

Los nodos de sintaxis son uno de los elementos principales de los árboles de sintaxis. Estos nodos representan construcciones sintácticas como declaraciones, instrucciones, cláusulas y expresiones. Cada categoría de nodos de sintaxis se representa mediante una clase independiente derivada de Microsoft.CodeAnalysis.SyntaxNode. El conjunto de clases de nodos no es extensible.

Todos los nodos de sintaxis son nodos no terminales del árbol de sintaxis, lo que significa que siempre tienen otros nodos y tokens como elementos secundarios. Como elemento secundario de otro nodo, cada nodo tiene un nodo principal al que se puede acceder mediante la propiedad SyntaxNode.Parent. Dado que los nodos y los árboles son inmutables, el elemento principal de un nodo nunca cambia. La raíz del árbol tiene un elemento principal nulo.

Cada nodo tiene un método SyntaxNode.ChildNodes() que devuelve una lista de nodos secundarios en orden secuencial según su posición en el texto de origen. Esta lista no contiene tokens. Cada nodo también tiene métodos para examinar descendientes, como DescendantNodes, DescendantTokens o DescendantTrivia, que representan una lista de todos los nodos, tokens o curiosidades que existen en el subárbol cuya raíz está en ese nodo.

Además, cada subclase de nodos de sintaxis expone los mismos elementos secundarios mediante propiedades fuertemente tipadas. Por ejemplo, una clase de nodos BinaryExpressionSyntax tiene tres propiedades adicionales específicas de los operadores binarios: Left, OperatorToken y Right. El tipo de Left y Right es ExpressionSyntax, y el tipo de OperatorToken es SyntaxToken.

Algunos nodos de sintaxis tienen elementos secundarios opcionales. Por ejemplo, IfStatementSyntax tiene un elemento opcional ElseClauseSyntax. Si el elemento secundario no está presente, la propiedad devuelve null.

Tokens de sintaxis

Los tokens de sintaxis son los terminales de la gramática del lenguaje y representan los fragmentos sintácticos más pequeños del código. Nunca son elementos principales de otros nodos o tokens. Los tokens de sintaxis constan de palabras clave, identificadores, literales y signos de puntuación.

Por eficacia, el tipo SyntaxToken es un tipo de valor CLR. Por tanto, a diferencia de los nodos de sintaxis, solo hay una estructura para todos los tipos de tokens con una mezcla de propiedades que tienen significado según el tipo de token que se va a representar.

Por ejemplo, un token de literal entero representa un valor numérico. Además del texto de origen sin formato que abarca el token, el token de literal tiene una propiedad Value que indica el valor entero descodificado exacto. El tipo de esta propiedad se considera Object, ya que puede ser alguno de los tipos primitivos.

La propiedad ValueText transmite la misma información que la propiedad Value; pero el tipo de esta propiedad siempre es String. Un identificador en el texto de origen C# puede incluir caracteres de escape Unicode, aunque la sintaxis de la propia secuencia de escape no se considera parte del nombre del identificador. Por tanto, aunque el texto sin formato que abarca el token incluye la secuencia de escape, la propiedad ValueText no. En su lugar, incluye los caracteres Unicode que identifica el escape. Por ejemplo, si el texto de origen contiene un identificador escrito como `\u03C0`, la propiedad ValueText de este token devuelve `π`.

Curiosidades de sintaxis

Las curiosidades de sintaxis representan las partes del texto de origen que no son realmente significativas para la correcta comprensión del código, como los espacios en blanco, los comentarios y las directivas de preprocesador. Al igual que los tokens de sintaxis, las curiosidades son tipos de valor. Se usa el tipo único Microsoft.CodeAnalysis.SyntaxTrivia para describir todos los tipos de curiosidades.

Dado que las curiosidades no forman parte de la sintaxis normal del lenguaje y pueden aparecer en cualquier lugar entre dos tokens, no se incluyen en el árbol de sintaxis como elemento secundario de un nodo. Pero dado que son importantes a la hora de implementar una característica como la refactorización y de mantener la plena fidelidad con el texto de origen, existen como parte del árbol de sintaxis.

Puede acceder a las curiosidades si inspecciona las colecciones SyntaxToken.LeadingTrivia o SyntaxToken.TrailingTrivia de un token. Cuando se analiza el texto de origen, las secuencias de curiosidades se asocian a los tokens. En general, un token es propietario de cualquier curiosidad que le preceda en la misma línea hasta el siguiente token. Cualquier curiosidad situada después de esa línea se asocia al token siguiente. El primer

token del archivo de origen obtiene todas las curiosidades iniciales, mientras que la última secuencia de curiosidades del archivo se agrega al último token del archivo, que, de lo contrario, tiene ancho de cero.

A diferencia de los nodos y los tokens de sintaxis, las curiosidades de sintaxis no tienen elementos principales. Pero, dado que forman parte del árbol y cada una está asociada a un token único, se puede acceder al token con el que está asociada mediante la propiedad SyntaxTrivia.Token.

Intervalos

Cada nodo, token o curiosidad conoce su posición dentro del texto de origen y el número de caracteres del que se compone. Una posición de texto se representa como un entero de 32 bits, que es un índice `char` de base cero. Un objeto TextSpan es la posición inicial y un recuento de caracteres, ambos representados como enteros. Si TextSpan tiene una longitud cero, hace referencia a una ubicación entre dos caracteres.

Cada nodo tiene dos propiedades TextSpan: Span y FullSpan.

La propiedad Span es el intervalo de texto desde el principio del primer token del subárbol del nodo al final del último token. Este intervalo no incluye ninguna curiosidad inicial ni final.

La propiedad FullSpan es el intervalo de texto que incluye el intervalo normal del nodo, así como el intervalo de cualquier curiosidad inicial o final.

Por ejemplo:

```
      if (x > 3)
      {
||        // this is bad
          |throw new Exception("Not right.");|  // better exception?||
      }
```

El nodo de la instrucción dentro del bloque tiene un intervalo indicado por las plecas (|). Incluye los caracteres `throw new Exception("Not right.");`. Las plecas dobles (||) indican el intervalo completo. Incluye los mismos caracteres que el intervalo y los caracteres asociados a las curiosidades inicial y final.

Tipos

Cada nodo, token o curiosidad tiene una propiedad SyntaxNode.RawKind, de tipo System.Int32, que identifica el elemento de sintaxis exacto representado. Este valor se puede convertir en una enumeración específica del lenguaje. Cada lenguaje, C# o Visual Basic, tiene una sola enumeración `SyntaxKind` (Microsoft.CodeAnalysis.CSharp.SyntaxKind y Microsoft.CodeAnalysis.VisualBasic.SyntaxKind, respectivamente) que enumera todos los posibles nodos, tokens y curiosidades de la gramática. Dicha conversión se puede realizar automáticamente. Para ello, es necesario acceder a los métodos de extensión CSharpExtensions.Kind o VisualBasicExtensions.Kind.

La propiedad RawKind permite anular fácilmente la ambigüedad de los tipos de nodos de sintaxis que comparten la misma clase de nodos. En el caso de los tokens y las curiosidades, esta propiedad es la única manera de distinguir un tipo de elemento de otro.

Por ejemplo, una sola clase BinaryExpressionSyntax tiene Left, OperatorToken y Right como elementos secundarios. La propiedad Kind distingue si es un tipo AddExpression, SubtractExpression o MultiplyExpression de nodo de sintaxis.

Errores

Aunque el texto de origen contenga errores de sintaxis, se expone un árbol de sintaxis completo con recorrido de ida y vuelta al origen. Si el analizador detecta código que no se ajusta a la sintaxis definida del lenguaje, usa una de

estas dos técnicas para crear un árbol de sintaxis.

- Si el analizador espera un determinado tipo de token, pero no lo encuentra, puede insertar un token que falta en el árbol de sintaxis en la ubicación esperada del token. Un token que falta representa el token real esperado, aunque tiene un intervalo vacío y su propiedad SyntaxNode.IsMissing devuelve `true`.
- El analizador puede omitir tokens hasta encontrar uno en el que pueda seguir analizando. En este caso, los tokens omitidos se adjuntan como un nodo de curiosidades con el tipo SkippedTokensTrivia.

Trabajar con semántica

18/03/2020 • 7 minutes to read • Edit Online

Los árboles de sintaxis representan la estructura léxica y sintáctica del código fuente. Aunque esta información por sí misma basta para describir todas las declaraciones y la lógica del origen, no es suficiente para identificar aquello a lo que se hace referencia. Un nombre puede representar:

- un tipo
- un campo
- un método
- una variable local

Aunque cada uno de ellos es exclusivamente diferente, determinar a cuál hace referencia realmente un identificador suele exigir una comprensión profunda de las reglas del lenguaje.

Hay elementos de programa representados en el código fuente y, además, los programas pueden hacer referencia a bibliotecas compiladas anteriormente, empaquetadas en archivos de ensamblado. Aunque no hay ningún código fuente y, por tanto, ningún nodo ni árbol de sintaxis, disponible para los ensamblados, los programas aún pueden hacer referencia a los elementos incluidos en ellos.

Para esas tareas necesita el **modelo semántico**.

Además de un modelo sintáctico del código fuente, un modelo semántico encapsula las reglas del lenguaje, lo que le ofrece una manera sencilla de combinar correctamente los identificadores con el elemento de programa correcto al que se hace referencia.

Compilación

Una compilación es una representación de todo lo necesario para compilar un programa de C# o Visual Basic, lo que incluye todas las referencias de ensamblado, las opciones de compilador y los archivos de origen.

Dado que toda esta información está en un solo lugar, los elementos incluidos en el código fuente pueden describirse con más detalle. La compilación representa cada tipo declarado, miembro o variable como un símbolo. La compilación contiene una serie de métodos que ayudan a encontrar y relacionar los símbolos que se han declarado en el código fuente o importado como metadatos desde un ensamblado.

Al igual que los árboles de sintaxis, las compilaciones son inmutables. Después de crear una compilación, ni el usuario ni nadie con quien la comparta puede modificarla. Pero puede crear una nueva compilación a partir de una existente, al especificar un cambio a medida que lo realiza. Por ejemplo, podría crear una compilación igual en todos los sentidos a una compilación existente, salvo que podría incluir un archivo de origen adicional o una referencia de ensamblado.

Símbolos

Un símbolo representa un elemento diferenciado declarado por el código fuente o importado desde un ensamblado como metadatos. Cada espacio de nombres, tipo, método, propiedad, campo, evento, parámetro o variable local se representa mediante un símbolo.

Una serie de métodos y propiedades del tipo Compilation ayudan a encontrar símbolos. Por ejemplo, puede buscar el símbolo de un tipo declarado por su nombre de metadatos común. También puede acceder a la tabla de símbolos completa como un árbol de símbolos enraizado por el espacio de nombres global.

Los símbolos también contienen información adicional que el compilador determina desde el origen o los metadatos, como otros símbolos referenciados. Cada tipo de símbolo se representa mediante una interfaz independiente derivada de ISymbol, cada una con sus propios métodos y propiedades que detallan la información recopilada por el compilador. Muchas de estas propiedades hacen referencia directamente a otros símbolos. Por ejemplo, la propiedad IMethodSymbol.ReturnType indica el símbolo de tipo real al que hace referencia la declaración del método.

Los símbolos presentan una representación común de espacios de nombres, tipos y miembros, entre el código fuente y los metadatos. Por ejemplo, un método que se ha declarado en el código fuente y un método que se ha importado desde los metadatos se representan mediante un elemento IMethodSymbol con las mismas propiedades.

Los símbolos son similares en concepto al sistema de tipos de CLR representado por la API System.Reflection, aunque son mejores en el aspecto de que modelan algo más que tipos. Los espacios de nombres, las variables locales y las etiquetas son todos símbolos. Además, los símbolos son una representación de conceptos del lenguaje, no de conceptos de CLR. Hay mucha superposición, pero también muchas distinciones significativas. Por ejemplo, un método Iterator de C# o Visual Basic es un único símbolo. Pero si el método Iterator se traduce a metadatos de CLR, es un tipo y varios métodos.

Modelo semántico

Un modelo semántico representa toda la información semántica de un solo archivo de origen. Puede usarlo para descubrir lo siguiente:

- Los símbolos a los que se hace referencia en una ubicación concreta del origen.
- El tipo resultante de cualquier expresión.
- Todos los diagnósticos, que son errores y advertencias.
- Cómo fluyen las variables hacia y desde las regiones del origen.
- Las respuestas a preguntas más especulativas.

Trabajar con un área de trabajo

18/03/2020 • 5 minutes to read • Edit Online

La capa **Áreas de trabajo** es el punto de partida para realizar análisis de código y refactorización de soluciones completas. En esta capa, la API de área de trabajo ayuda a organizar toda la información sobre los proyectos de una solución en un modelo de objetos único, lo que ofrece acceso directo a modelos de objetos de capa de compilador como texto de origen, árboles de sintaxis, modelos semánticos y compilaciones sin necesidad de analizar archivos, configurar opciones ni administrar dependencias entre proyectos.

Los entornos de host, como un IDE, proporcionan un área de trabajo que corresponde a la solución abierta. También es posible usar este modelo fuera de un IDE con solo cargar un archivo de solución.

Área de trabajo

Un área de trabajo es una representación activa de la solución como una colección de proyectos, cada uno con una colección de documentos. Normalmente, un área de trabajo está asociada a un entorno de host en continuo cambio a medida que el usuario escribe o manipula las propiedades.

Workspace proporciona acceso al modelo actual de la solución. Cuando se produce un cambio en el entorno de host, el área de trabajo desencadena los eventos correspondientes y la propiedad Workspace.CurrentSolution se actualiza. Por ejemplo, cuando el usuario escribe en un editor de texto correspondiente a uno de los documentos de origen, el área de trabajo usa un evento para indicar que ha cambiado el modelo general de la solución y qué documento se ha modificado. Luego puede reaccionar a esos cambios mediante el análisis de la corrección del nuevo modelo, resaltando las áreas de importancia o realizando una sugerencia para un cambio de código.

También puede crear áreas de trabajo independientes desconectadas del entorno de host o que se usen en una aplicación sin entorno de host.

Soluciones, proyectos, documentos

Aunque un área de trabajo puede cambiar cada vez que se pulsa una tecla, puede trabajar con el modelo de la solución de forma aislada.

Una solución es un modelo inmutable de los proyectos y los documentos. Esto significa que el modelo se puede compartir sin que haya bloqueo ni duplicación. Después de obtener una instancia de solución de la propiedad Workspace.CurrentSolution, esa instancia no cambia nunca. Pero, al igual que con los árboles de sintaxis y las compilaciones, puede modificar soluciones mediante la creación de nuevas instancias basadas en soluciones existentes y cambios concretos. Para que el área de trabajo refleje los cambios, debe aplicar explícitamente la solución modificada al área de trabajo.

Un proyecto es una parte del modelo de solución general inmutable. Representa todos los documentos de código de origen, las opciones de análisis y compilación, y las referencias de ensamblado y de proyecto a proyecto. Desde un proyecto puede acceder a la compilación correspondiente sin necesidad de determinar las dependencias del proyecto ni de analizar los archivos de origen.

Un documento también es una parte del modelo de solución general inmutable. Un documento representa un solo archivo de origen desde el que se puede acceder al texto del archivo, el árbol de sintaxis y el modelo semántico.

El diagrama siguiente es una representación de la relación entre el área de trabajo y el entorno de host, las herramientas y cómo se realizan las modificaciones.

Resumen

Roslyn expone un conjunto de API de compilador y API de áreas de trabajo que proporciona información detallada sobre el código fuente y que tiene plena fidelidad con los lenguajes C# y Visual Basic. El SDK de .NET Compiler Platform reduce notablemente la barrera para crear herramientas y aplicaciones centradas en el código. Crea numerosas oportunidades para la innovación en áreas como la metaprogramación, la generación y la transformación de código, el uso interactivo de los lenguajes C# y Visual Basic, y la inserción de C# y Visual Basic en lenguajes específicos del dominio.

Explorar código con el Visualizador de sintaxis Roslyn en Visual Studio

13/05/2020 • 19 minutes to read • Edit Online

En este artículo se ofrece información general de la herramienta Visualizador de sintaxis que se incluye como parte del SDK de .NET Compiler Platform ("Roslyn"). El Visualizador de sintaxis es una ventana de herramientas con la que puede inspeccionar y explorar árboles de sintaxis. Es una herramienta esencial para comprender los modelos de código que quiere analizar. También es útil para la depuración al desarrollar sus propias aplicaciones con el SDK de .NET Compiler Platform ("Roslyn"). Abra esta herramienta cuando vaya a crear sus primeros analizadores. Con el visualizador comprenderá los modelos usados por las API. También puede usar herramientas como SharpLab o LINQPad para inspeccionar el código y comprender los árboles de sintaxis.

Instrucciones de instalación: Instalador de Visual Studio

Hay dos maneras distintas de buscar el **SDK de .NET Compiler Platform** en el **Instalador de Visual Studio**:

Instalación con el Instalador de Visual Studio: visualización de cargas de trabajo

El SDK de .NET Compiler Platform no se selecciona automáticamente como parte de la carga de trabajo de desarrollo de extensiones de Visual Studio. Se debe seleccionar como un componente opcional.

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Active la carga de trabajo **Desarrollo de extensiones de Visual Studio**.
4. Abra el nodo **Desarrollo de extensiones de Visual Studio** en el árbol de resumen.
5. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en última posición bajo los componentes opcionales.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Abra el nodo **Componentes individuales** en el árbol de resumen.
2. Active la casilla **Editor de DGML**.

Instalación con el Instalador de Visual Studio: pestaña Componentes individuales

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Haga clic en la pestaña **Componentes individuales**.
4. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en la parte superior bajo la sección **Compiladores, herramientas de compilación y tiempos de ejecución**.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Active la casilla **Editor de DGML**. La encontrará en la sección **Herramientas de código**.

Para familiarizarse con los conceptos usados en el SDK de .NET Compiler Platform, lea el artículo introductorio. Proporciona una introducción a los árboles de sintaxis, los nodos, los tokens y algunas curiosidades.

Visualizador de sintaxis

Syntax Visualizer permite inspeccionar el árbol de sintaxis del archivo de código de C# o Visual Basic en la ventana del editor activo actual en el IDE de Visual Studio. El visualizador se puede iniciar haciendo clic en **Vista** >

Other Windows (Otras ventanas) > **Syntax Visualizer (Visualizador de sintaxis)** . También puede usar la barra de herramientas **Inicio rápido** en la esquina superior derecha. Escriba "syntax" y se mostrará el comando para abrir el **Visualizador de sintaxis**.

Este comando abre el Visualizador de sintaxis como una ventana de herramientas flotante. Si no tiene abierta ninguna ventana de editor de código, la presentación está en blanco, tal como se muestra en esta imagen.

Acople esta ventana de herramienta en una ubicación cómoda dentro de Visual Studio, como por ejemplo, en el lado izquierdo. El visualizador muestra información sobre el archivo de código actual.

Cree un nuevo proyecto con los comandos **Archivo** > **Nuevo proyecto**. Puede crear un proyecto de Visual Basic o C#. Cuando Visual Studio abre el principal archivo de código para este proyecto, el visualizador muestra el árbol de sintaxis correspondiente. Puede abrir cualquier archivo de C# o Visual Basic existente en esta instancia de Visual Studio y el visualizador mostrará el árbol de sintaxis de ese archivo. Si tiene varios archivos de código abiertos dentro de Visual Studio, el visualizador muestra el árbol de sintaxis para el archivo de código activo (el archivo de código que tiene el foco de teclado).

- C#
- Visual Basic

Como se muestra en las imágenes anteriores, la ventana de herramientas del visualizador muestra el árbol de sintaxis en la parte superior y una cuadrícula de propiedades en la parte inferior. La cuadrícula de propiedades muestra las propiedades del elemento que está seleccionado actualmente en el árbol, incluido el *Tipo* de .NET y la *Variante* (SyntaxKind) del elemento.

Los árboles de sintaxis incluyen tres tipos de elementos: *nodos*, *tokens* y *curiosidades*. Encontrará más información sobre estos tipos en el artículo Trabajar con sintaxis. Los elementos de cada tipo se representan mediante un color diferente. Haga clic en el botón "Leyenda" para saber más sobre los colores usados.

Cada elemento del árbol también muestra su propio **intervalo**. El **intervalo** está comprendido por los índices (la posición inicial y la final) de ese nodo en el archivo de texto. En el anterior ejemplo de C#, el token "UsingKeyword [0..5)" seleccionado tiene un **intervalo** de cinco caracteres de ancho [0..5). La notación "[.)" significa que el índice inicial forma parte del intervalo, pero el índice final no.

Se puede navegar por el árbol de dos maneras:

- Expandir el árbol o hacer clic en él. El visualizador selecciona automáticamente el texto correspondiente al intervalo de este elemento en el editor de código.
- Hacer clic en el texto o seleccionarlo en el editor de código. En el ejemplo de Visual Basic anterior, si selecciona la línea que contiene "Module Module1" en el editor de código, el visualizador navega automáticamente al nodo ModuleStatement correspondiente en el árbol.

El visualizador resalta el elemento en el árbol cuyo intervalo coincide mejor con el intervalo del texto seleccionado en el editor.

El visualizador actualiza el árbol para coincidir con las modificaciones en el archivo de código activo. Agregue una llamada a `Console.WriteLine()` dentro de `Main()`. A medida que escribe, el visualizador actualiza el árbol.

Pare de escribir en cuanto escriba `Console.`. Verá que el árbol ha marcado en rosa algunos elementos. En este momento, hay errores (también denominados "diagnósticos") en el código escrito. Estos errores se adjuntan a los nodos, los tokens y las curiosidades en el árbol de sintaxis. El visualizador muestra qué elementos tienen errores adjuntados a ellos resaltando el fondo en color rosa. Puede inspeccionar los errores en cualquier elemento marcado en rosa si desplaza el puntero sobre el elemento. El visualizador muestra solo los errores sintácticos (los errores relacionados con la sintaxis del código escrito) y no muestra los errores semánticos.

Gráficos de sintaxis

Haga clic con el botón derecho en cualquier elemento del árbol y haga clic en **View Directed Syntax Graph** (Ver gráfico de sintaxis dirigido).

- C#
- Visual Basic

El visualizador muestra una representación gráfica del subárbol cuya raíz se encuentra en el elemento seleccionado. Siga estos pasos para el nodo **MethodDeclaration** correspondiente al método `Main()` en el ejemplo de C#. El visualizador muestra un gráfico de sintaxis que tiene este aspecto:

El visor de gráficos de sintaxis tiene una opción para mostrar una leyenda para su esquema de color. También puede pasar el puntero sobre determinados elementos en el gráfico de sintaxis para ver las propiedades correspondientes a ese elemento.

Puede ver gráficos de sintaxis de elementos diferentes en el árbol de manera repetida y los gráficos siempre se mostrarán en la misma ventana dentro de Visual Studio. Puede acoplar esta ventana en una ubicación cómoda en Visual Studio para no tener que cambiar entre pestañas para ver un nuevo gráfico de sintaxis. La parte inferior, debajo de las ventanas del editor de código, suele resultar cómoda.

Este es el diseño de acoplamiento que se usa con la ventana del visualizador y la ventana del gráfico de sintaxis:

Otra opción consiste en colocar la ventana del gráfico de sintaxis en un segundo monitor, en una configuración de monitor dual.

Semántica de inspección

El Visualizador de sintaxis permite realizar una inspección rudimentaria de símbolos e información semántica. Escriba `double x = 1 + 1;` dentro de Main() en el ejemplo de C#. Después, seleccione la expresión `1 + 1` en la ventana del editor de código. El visualizador resalta el nodo **AddExpression** en el visualizador. Haga clic con el botón derecho en **AddExpression** y elija **View Symbol (if any)** [Ver símbolo (si existe)]. Tenga en cuenta que la mayoría de los elementos de menú tienen el calificador "si existe". El Visualizador de sintaxis inspecciona las propiedades de un nodo, incluidas las propiedades que es posible que no estén presentes para todos los nodos.

La cuadrícula de propiedades del visualizador se actualiza tal como se muestra en la figura siguiente: El símbolo de la expresión es un símbolo **SynthesizedIntrinsicOperatorSymbol** con **Kind = Method**.

Intente **View TypeSymbol (if any)** [Ver TypeSymbol (si existe)] para el mismo nodo **AddExpression**. La cuadrícula de propiedades del visualizador se actualiza como se muestra en esta imagen, que indica que el tipo de la expresión seleccionada es `Int32`.

Intente **View Converted TypeSymbol (if any)** [Ver TypeSymbol convertido (si existe)] para el mismo nodo **AddExpression**. La cuadrícula de propiedades se actualiza para indicar que, aunque el tipo de la expresión es `Int32`, el tipo convertido de la expresión es `Double`, como se muestra en esta imagen. Este nodo incluye información de símbolo de tipo convertido porque la expresión `Int32` se produce en un contexto donde se debe convertir a `Double`. Esta conversión satisface el tipo `Double` especificado para la variable `x` en el lado izquierdo del operador de asignación.

Por último, intente **View Constant Value (if any)** [Ver valor de constante (si existe)] para el mismo nodo **AddExpression**. La cuadrícula de propiedades muestra que el valor de la expresión es una constante en tiempo de compilación con el valor `2`.

El ejemplo anterior también se puede replicar en Visual Basic. Escriba `Dim x As Double = 1 + 1` en un archivo de Visual Basic. Seleccione la expresión `1 + 1` en la ventana del editor de código. El visualizador resalta el nodo **AddExpression** correspondiente en el visualizador. Repita los pasos anteriores para **AddExpression** y deberían

mostrarse resultados idénticos.

Examine más código en Visual Basic. Actualice el archivo principal de Visual Basic con este código:

```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

Este código incluye un alias llamado `C` que se asigna al tipo `System.Console` en la parte superior del archivo y usa este alias en `Main()`. Seleccione el uso de este alias, `C` en `C.WriteLine()`, dentro del método `Main()`. El visualizador selecciona el nodo **IdentifierName** correspondiente en el visualizador. Haga clic con el botón derecho en este nodo y elija **View Symbol (if any)** [Ver símbolo (si existe)]. La cuadrícula de propiedades indica que este identificador está enlazado al tipo `System.Console` tal como se muestra en esta imagen:

Intente **View AliasSymbol (if any)** [Ver AliasSymbol (si existe)] para el mismo nodo **IdentifierName**. La cuadrícula de propiedades indica que el identificador es un alias con el nombre `C` que está enlazado al destino `System.Console`. En otras palabras, la cuadrícula de propiedades proporciona información sobre el **AliasSymbol** correspondiente al identificador `C`.

Inspeccione el símbolo correspondiente a cualquier tipo, método o propiedad declarados. Seleccione el nodo correspondiente en el visualizador y haga clic en **View Symbol (if any)** [Ver símbolo (si existe)]. Seleccione el método `Sub Main()`, incluido el cuerpo del método. Haga clic en **View Symbol (if any)** [Ver símbolo (si existe)] para el nodo **SubBlock** correspondiente en el visualizador. La cuadrícula de propiedades muestra que **MethodSymbol** para este nodo **SubBlock** tiene el nombre `Main` con el tipo de valor devuelto `Void`.

Los ejemplos de Visual Basic anteriores se pueden replicar fácilmente en C#. Escriba `using C = System.Console;` en lugar de `Imports C = System.Console` para el alias. Los pasos anteriores en C# producen resultados idénticos en la ventana del visualizador.

Las operaciones de inspección semántica solo están disponibles en los nodos. No están disponibles en tokens o curiosidades. No todos los nodos tienen información semántica interesante que inspeccionar. Cuando un nodo no tiene información semántica interesante, al hacer clic en **View * Symbol (if any)** [Ver símbolo (si existe)] se muestra una cuadrícula de propiedades en blanco.

Puede leer más sobre las API para realizar análisis semánticos en el documento introductorio Trabajar con semántica.

Cerrar el Visualizador de sintaxis

Puede cerrar la ventana del visualizador cuando no esté usándolo para examinar el código fuente. Syntax Visualizer se actualiza a medida que navega por el código, y modifica o cambia el código fuente. Puede distraerse cuando no esté usándolo.

Introducción al análisis de sintaxis

18/03/2020 • 25 minutes to read • Edit Online

En este tutorial, explorará la **API de sintaxis**. La API de sintaxis proporciona acceso a las estructuras de datos que describen un programa de C# o Visual Basic. Estas estructuras de datos tienen suficientes detalles para representar completamente un programa de cualquier tamaño. Estas estructuras pueden describir programas completos que se compilen y ejecuten correctamente. También pueden describir programas incompletos, conforme los escribe, en el editor.

Para habilitar esta expresión completa, las estructuras de datos y las API que constituyen la API de sintaxis son necesariamente complejas. Empecemos con el aspecto de la estructura de datos para el programa típico "Hola mundo":

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Mire el texto del programa anterior. Reconoce elementos conocidos. Todo el texto representa un único archivo de código fuente o una **unidad de compilación**. Las tres primeras líneas del archivo de código fuente son **directivas using**. El código fuente restante se encuentra en una **declaración de espacio de nombres**. La declaración de espacio de nombres contiene una **declaración de clase** secundaria. La declaración de clase contiene una **declaración de método**.

La API de sintaxis crea una estructura de árbol y la raíz representa la unidad de compilación. Los nodos del árbol representan las directivas using, la declaración del espacio de nombres y todos los demás elementos del programa. La estructura de árbol continúa hasta los niveles inferiores: la cadena "Hello World!" es un **token de literal de cadena** que es un descendiente de un **argumento**. La API de sintaxis proporciona acceso a la estructura del programa. Puede consultar los procedimientos de código concretos, recorrer el árbol completo para entender el código y crear árboles al modificar el árbol existente.

Esa descripción breve proporciona información general sobre el tipo de información accesible mediante la API de sintaxis. La API de sintaxis no es nada más que una API formal que describe las construcciones de código que ya conoce de C#. Entre las funcionalidades completas, se incluye información sobre cómo se da formato al código, incluidos los saltos de línea, los espacios en blanco y la sangría. Con esta información, puede representar por completo el código tal y como lo escriben y leen los programadores humanos o el compilador. Con esta estructura, puede interactuar con el código fuente de forma muy significativa. Ya no son cadenas de texto, sino datos que representan la estructura de un programa de C#.

Para empezar, debe instalar el **SDK de .NET Compiler Platform**:

Instrucciones de instalación: Instalador de Visual Studio

Hay dos maneras distintas de buscar el **SDK de .NET Compiler Platform** en el **Instalador de Visual Studio**:

Instalación con el Instalador de Visual Studio: visualización de cargas de trabajo

El SDK de .NET Compiler Platform no se selecciona automáticamente como parte de la carga de trabajo de desarrollo de extensiones de Visual Studio. Se debe seleccionar como un componente opcional.

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Active la carga de trabajo **Desarrollo de extensiones de Visual Studio**.
4. Abra el nodo **Desarrollo de extensiones de Visual Studio** en el árbol de resumen.
5. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en última posición bajo los componentes opcionales.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Abra el nodo **Componentes individuales** en el árbol de resumen.
2. Active la casilla **Editor de DGML**.

Instalación con el Instalador de Visual Studio: pestaña Componentes individuales

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Haga clic en la pestaña **Componentes individuales**.
4. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en la parte superior bajo la sección **Compiladores, herramientas de compilación y tiempos de ejecución**.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Active la casilla **Editor de DGML**. La encontrará en la sección **Herramientas de código**.

Comprender los árboles de sintaxis

Use la API de sintaxis para cualquier análisis de la estructura del código de C#. La **API de sintaxis** expone los analizadores, los árboles de sintaxis y las utilidades para analizar y construir árboles de sintaxis. Es la forma en que busca en el código cualquier elemento de sintaxis específica o lee el código de un programa.

Un árbol de sintaxis es una estructura de datos que usan los compiladores de C# y Visual Basic para comprender los programas de C# y Visual Basic. Los árboles de sintaxis los produce el mismo analizador que se ejecuta cuando se compila un proyecto o un programador presiona F5. Los árboles de sintaxis tienen una fidelidad completa con el lenguaje; cada bit de información en un archivo de código se representa en el árbol. Escribir un árbol de sintaxis en texto reproduce el texto original exacto que se ha analizado. Los árboles de sintaxis también son **inmutables**; una vez creado un árbol de sintaxis, nunca se puede modificar. Los consumidores de los árboles pueden analizarlos en varios subprocesos, sin bloqueos ni otras medidas de simultaneidad, sabiendo que los datos nunca cambian. Puede usar las API para crear árboles que sean el resultado de modificar un árbol existente.

Los cuatro pilares principales de los árboles de sintaxis son los siguientes:

- La clase Microsoft.CodeAnalysis.SyntaxTree, una instancia de lo que representa un árbol de análisis completo. SyntaxTree es una clase abstracta que tiene derivados específicos del lenguaje. Use los métodos de análisis de la clase Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree (o Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree) para analizar texto en C# o Visual Basic.
- La clase Microsoft.CodeAnalysis.SyntaxNode, instancias de lo que representan las construcciones sintácticas como declaraciones, instrucciones, cláusulas y expresiones.
- La estructura Microsoft.CodeAnalysis.SyntaxToken, que representa una palabra clave, identificador, operador o puntuación individuales.
- Por último, la estructura Microsoft.CodeAnalysis.SyntaxTrivia, que representa sintácticamente bits de

información insignificante, como el espacio en blanco entre tokens, las directivas de preprocesamiento y los comentarios.

La trivialidad, los tokens y los nodos se componen de forma jerárquica para formar un árbol que representa por completo todo lo que hay en un fragmento de código de Visual Basic o C#. Puede ver esta estructura mediante la ventana **Syntax Visualizer** (Visualizador de sintaxis). En Visual Studio, elija **Vista** > **Otras ventanas** > **Syntax Visualizer** (Visualizador de sintaxis). Por ejemplo, el archivo de código fuente de C# anterior examinado con **Syntax Visualizer** (Visualizador de sintaxis) tiene el mismo aspecto que en la siguiente ilustración:

SyntaxNode: azul | **SyntaxToken**: verde | **SyntaxTrivia**: rojo

Si se desplaza por esta estructura de árbol, podrá encontrar cualquier instrucción, expresión, token o bit de espacio en blanco en un archivo de código.

Aunque puede buscar cualquier elemento en un archivo de código mediante las API de sintaxis, la mayoría de los escenarios implican examinar pequeños fragmentos de código o buscar instrucciones o fragmentos concretos. Los dos ejemplos siguientes muestran usos típicos para examinar la estructura del código o buscar instrucciones únicas.

Recorrer árboles

Puede examinar los nodos de un árbol de sintaxis de dos maneras. Puede recorrer el árbol para examinar cada

nodo o puede consultar elementos o nodos concretos.

Recorrido manual

Puede ver el código terminado de este ejemplo en nuestro repositorio de GitHub.

> **NOTE**
> Los tipos de árbol de sintaxis usan la herencia para describir los diferentes elementos de sintaxis que son válidos en diferentes ubicaciones del programa. A menudo, usar estas API significa convertir propiedades o miembros de colección en tipos derivados concretos. En los ejemplos siguientes, la asignación y las conversiones son instrucciones independientes, con variables con tipo explícito. Puede leer el código para ver los tipos de valor devuelto de la API y el tipo de motor de ejecución de los objetos devueltos. En la práctica, es más habitual usar variables con tipo implícito y basarse en nombres de API para describir el tipo de los objetos que se examinan.

Cree un proyecto de **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente) de C#:

- En Visual Studio, elija **Archivo** > **Nuevo** > **Proyecto** para mostrar el cuadro de diálogo Nuevo proyecto.
- En **Visual C#** > **Extensibilidad**, elija **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente).
- Asigne al proyecto el nombre "**SyntaxTreeManualTraversal**" y haga clic en Aceptar.

Va a analizar el programa básico "Hola mundo" mostrado anteriormente. Agregue el texto para el programa Hola mundo como una constante en su clase `Program`:

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

A continuación, agregue el código siguiente para crear el **árbol de sintaxis** para el texto del código de la constante `programText`. Agregue la línea siguiente al método `Main`:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Estas dos líneas crean el árbol y recuperan su nodo raíz. Ahora puede examinar los nodos del árbol. Agregue estas líneas al método `Main` para mostrar algunas de las propiedades del nodo raíz en el árbol:

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");
```

Ejecute la aplicación para ver lo que ha detectado el código sobre el nodo raíz de este árbol.

Normalmente, recorrería el árbol para obtener información sobre el código. En este ejemplo, analiza código que conoce para explorar las API. Agregue el código siguiente para examinar el primer miembro del nodo `root`:

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

Ese miembro es una Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax. Representa todo lo que se incluye en el ámbito de la declaración `namespace HelloWorld`. Agregue el código siguiente para examinar qué nodos se declaran en el espacio de nombres `HelloWorld`:

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared in this namespace.");
WriteLine($"The first member is a {helloWorldDeclaration.Members[0].Kind()}.");
```

Ejecute el programa para ver lo que ha aprendido.

Ahora que sabe que la declaración es una Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax, declare una nueva variable de ese tipo para examinar la declaración de clase. Esta clase solo contiene un miembro: el método `Main`. Agregue el código siguiente para buscar el método `Main` y conviértalo en una Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax.

```
var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in the
{programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];
```

El nodo de declaración de método contiene toda la información sintáctica sobre el método. Vamos a mostrar el tipo de valor devuelto del método `Main`, el número y los tipos de los argumentos, y el texto del cuerpo del método. Agregue el código siguiente:

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is {mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count} parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is {item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method follows:");
WriteLine(mainDeclaration.Body.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Ejecute el programa para ver toda la información que ya conoce sobre este programa:

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using statements. They are:
        System
        System.Collections
        System.Linq
        System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
        {
            Console.WriteLine("Hello, World!");
        }
```

Métodos de consulta

Además de recorrer árboles, también puede explorar el árbol de sintaxis mediante los métodos de consulta definidos en Microsoft.CodeAnalysis.SyntaxNode. Cualquier persona que conozca XPath debería conocer estos métodos. Puede usarlos con LINQ para buscar elementos rápidamente en un árbol. SyntaxNode tiene métodos de consulta como DescendantNodes, AncestorsAndSelf y ChildNodes.

Puede usar estos métodos de consulta para buscar el argumento para el método `Main` como una alternativa a navegar por el árbol. Agregue el siguiente código en la parte inferior del método `Main`:

```
var firstParameters = from methodDeclaration in root.DescendantNodes()
                                        .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);
```

La primera instrucción usa una expresión LINQ y el método DescendantNodes para buscar el mismo parámetro que en el ejemplo anterior.

Ejecute el programa y compruebe que la expresión LINQ ha encontrado el mismo parámetro que se encuentra al navegar por el árbol de forma manual.

En el ejemplo, se usan instrucciones `WriteLine` para mostrar información sobre los árboles de sintaxis conforme se recorren. Puede obtener mucha más información si ejecuta el programa terminado en el depurador. Puede examinar más propiedades y métodos que forman parte del árbol de sintaxis creado para el programa Hola mundo.

Rastreadores de sintaxis

A menudo, quiere buscar todos los nodos de un tipo concreto en un árbol de sintaxis, por ejemplo, todas las declaraciones de propiedad de un archivo. Si extiende la clase Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker e invalida el método VisitPropertyDeclaration(PropertyDeclarationSyntax), se procesan todas las declaraciones de propiedad de un árbol de sintaxis sin conocer su estructura de antemano. CSharpSyntaxWalker es un tipo determinado de CSharpSyntaxVisitor que visita de forma recurrente un nodo y todos sus elementos secundarios.

En este ejemplo, se implementa un CSharpSyntaxWalker que examina un árbol de sintaxis. Recopila directivas

`using` que determina que no implementan un espacio de nombres `System`.

Cree un proyecto de **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente) de C# y asígnele el nombre "**SyntaxWalker**".

Puede ver el código terminado de este ejemplo en nuestro repositorio de GitHub. El ejemplo de GitHub contiene los dos proyectos que se describen en este tutorial.

Como en el ejemplo anterior, puede definir una constante de cadena para que contenga el texto del programa que se va a analizar:

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
    }
}";
```

Este texto de origen contiene directivas `using` dispersas por cuatro ubicaciones diferentes: el nivel de archivo, en el espacio de nombres de nivel superior y en los dos espacios de nombres anidados. En este ejemplo, se destaca un escenario principal para usar la clase CSharpSyntaxWalker en el código de la consulta. Sería complejo visitar todos los nodos del árbol de sintaxis raíz para buscar las declaraciones using. En su lugar, cree una clase derivada y reemplace el método al que se llama solo cuando el nodo actual del árbol sea una directiva using. El visitante no hace ningún trabajo en ningún otro tipo de nodo. Este método único examina todas las instrucciones `using` y compila una colección de los espacios de nombres que no están en el espacio de nombres `System`. Compile un CSharpSyntaxWalker que examine todas las instrucciones `using`, pero solo las instrucciones `using`.

Ahora que ha definido el texto del programa, debe crear un `SyntaxTree` y obtener la raíz de ese árbol:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

A continuación, cree una clase. En Visual Studio, elija **Proyecto** > **Agregar nuevo elemento**. En el cuadro de diálogo **Agregar nuevo elemento**, escriba *UsingCollector.cs* como nombre de archivo.

Implemente la funcionalidad del visitante `using` en la clase `UsingCollector`. Para empezar, haga que la clase `UsingCollector` derive de CSharpSyntaxWalker.

```
class UsingCollector : CSharpSyntaxWalker
```

Necesita almacenamiento para contener los nodos del espacio de nombres que está recopilando. Declare una propiedad pública de solo lectura en la clase `UsingCollector`; use esta variable para almacenar los nodos UsingDirectiveSyntax que encuentre:

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new List<UsingDirectiveSyntax>();
```

La clase base, CSharpSyntaxWalker, implementa la lógica para visitar todos los nodos del árbol de sintaxis. La clase derivada reemplaza los métodos llamados por los nodos específicos que le interesan. En este caso, le interesa cualquier directiva `using`. Por tanto, debe invalidar el método VisitUsingDirective(UsingDirectiveSyntax). El único argumento de este método es un objeto Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax. Se trata de una ventaja importante de usar los visitantes: llaman a los métodos invalidados con argumentos que ya se han convertido al tipo de nodo concreto. La clase Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax tiene una propiedad Name que almacena el nombre del espacio de nombres que se va a importar. Es una Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax. Agregue el código siguiente en la invalidación VisitUsingDirective(UsingDirectiveSyntax):

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\t\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

Como con el ejemplo anterior, ha agregado una variedad de instrucciones `WriteLine` para ayudar a comprender este método. Puede ver cuándo se llama y qué argumentos se le pasan cada vez.

Por último, debe agregar dos líneas de código para crear el `UsingCollector` y hacer que visite el nodo raíz y recopile todas las instrucciones `using`. A continuación, agregue un bucle `foreach` para que muestre todas las instrucciones `using` que encuentre el recopilador:

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Compile y ejecute el programa. Debería ver los siguientes resultados:

```
        VisitUsingDirective called with System.
        VisitUsingDirective called with System.Collections.Generic.
        VisitUsingDirective called with System.Linq.
        VisitUsingDirective called with System.Text.
        VisitUsingDirective called with Microsoft.CodeAnalysis.
                Success. Adding Microsoft.CodeAnalysis.
        VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
                Success. Adding Microsoft.CodeAnalysis.CSharp.
        VisitUsingDirective called with Microsoft.
                Success. Adding Microsoft.
        VisitUsingDirective called with System.ComponentModel.
        VisitUsingDirective called with Microsoft.Win32.
                Success. Adding Microsoft.Win32.
        VisitUsingDirective called with System.Runtime.InteropServices.
        VisitUsingDirective called with System.CodeDom.
        VisitUsingDirective called with Microsoft.CSharp.
                Success. Adding Microsoft.CSharp.
Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

¡Enhorabuena! Ha usado la **API de sintaxis** para buscar tipos concretos de instrucciones y declaraciones de C# en el código fuente de C#.

Introducción al análisis semántico

18/03/2020 • 16 minutes to read • Edit Online

En este tutorial, se asume que conoce la API de sintaxis. En el artículo Introducción al análisis de sintaxis se proporciona una introducción suficiente.

En este tutorial, explorará las **API de símbolo** y **enlace**. Estas API ofrecen información sobre el *significado semántico* de un programa. Le permiten formular y responder preguntas sobre los tipos representados por cualquier símbolo en el programa.

Deberá instalar el **SDK de .NET Compiler Platform**:

Instrucciones de instalación: Instalador de Visual Studio

Hay dos maneras distintas de buscar el **SDK de .NET Compiler Platform** en el **Instalador de Visual Studio**:

Instalación con el Instalador de Visual Studio: visualización de cargas de trabajo

El SDK de .NET Compiler Platform no se selecciona automáticamente como parte de la carga de trabajo de desarrollo de extensiones de Visual Studio. Se debe seleccionar como un componente opcional.

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Active la carga de trabajo **Desarrollo de extensiones de Visual Studio**.
4. Abra el nodo **Desarrollo de extensiones de Visual Studio** en el árbol de resumen.
5. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en última posición bajo los componentes opcionales.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Abra el nodo **Componentes individuales** en el árbol de resumen.
2. Active la casilla **Editor de DGML**.

Instalación con el Instalador de Visual Studio: pestaña Componentes individuales

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Haga clic en la pestaña **Componentes individuales**.
4. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en la parte superior bajo la sección **Compiladores, herramientas de compilación y tiempos de ejecución**.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Active la casilla **Editor de DGML**. La encontrará en la sección **Herramientas de código**.

Comprender las compilaciones y los símbolos

Conforme trabaja más con el SDK de .NET Compiler, empieza a familiarizarse con las diferencias entre la API de sintaxis y la API semántica. La **API de sintaxis** le permite buscar en la *estructura* de un programa. En cambio, a menudo quiere una información más completa sobre la semántica o el *significado* de un programa. Aunque un fragmento de código o archivo de código dinámico de Visual Basic o C# se puede analizar sintácticamente de forma aislada, no tiene sentido formular preguntas como "¿cuál es el tipo de esta variable?" de manera unilateral. El significado de un nombre de tipo puede depender de las referencias de ensamblado, las importaciones de

espacio de nombres u otros archivos de código. Esas preguntas se responden mediante la **API semántica**, concretamente la clase Microsoft.CodeAnalysis.Compilation.

Una instancia de Compilation es análoga a un único proyecto, tal como muestra el compilador y representa todo lo necesario para compilar un programa de Visual Basic o C#. La **compilación** incluye el conjunto de archivos de código fuente que se compilarán, las referencias de ensamblado y las opciones del compilador. Puede analizar el significado del código con toda la demás información en este contexto. Una Compilation permite buscar **símbolos** (entidades como tipos, espacios de nombres, miembros y variables a los que hacen referencia nombres y otras expresiones). El proceso de asociar los nombres y las expresiones con **símbolos** se denomina **enlace**.

Como Microsoft.CodeAnalysis.SyntaxTree, Compilation es una clase abstracta con derivados específicos del lenguaje. Al crear una instancia de la compilación, debe invocar un método de generador en la clase Microsoft.CodeAnalysis.CSharp.CSharpCompilation (o Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation).

Consultar símbolos

En este tutorial, volverá a examinar el programa "Hola mundo". En esta ocasión, consultará los símbolos del programa para comprender qué tipos representan esos símbolos. Consultará los tipos en un espacio de nombres y aprenderá a buscar los métodos disponibles en un tipo.

Puede ver el código terminado de este ejemplo en nuestro repositorio de GitHub.

> **NOTE**
> Los tipos de árbol de sintaxis usan la herencia para describir los diferentes elementos de sintaxis que son válidos en diferentes ubicaciones del programa. A menudo, usar estas API significa convertir propiedades o miembros de colección en tipos derivados concretos. En los ejemplos siguientes, la asignación y las conversiones son instrucciones independientes, con variables con tipo explícito. Puede leer el código para ver los tipos de valor devuelto de la API y el tipo de motor de ejecución de los objetos devueltos. En la práctica, es más habitual usar variables con tipo implícito y basarse en nombres de API para describir el tipo de los objetos que se examinan.

Cree un proyecto de **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente) de C#:

- En Visual Studio, elija **Archivo** > **Nuevo** > **Proyecto** para mostrar el cuadro de diálogo Nuevo proyecto.
- En **Visual C#** > **Extensibilidad**, elija **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente).
- Asigne al proyecto el nombre "**SemanticQuickStart**" y haga clic en Aceptar.

Va a analizar el programa básico "Hola mundo" mostrado anteriormente. Agregue el texto para el programa Hola mundo como una constante en su clase `Program`:

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

A continuación, agregue el código siguiente para crear el árbol de sintaxis para el texto del código de la constante `programText`. Agregue la línea siguiente al método `Main`:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

A continuación, compile una CSharpCompilation del árbol que ya ha creado. El ejemplo "Hola mundo" se basa en los tipos String y Console. Debe hacer referencia al ensamblado que declara esos dos tipos en la compilación. Agregue la siguiente línea a su método `Main` para crear una compilación de su árbol de sintaxis, incluida la referencia al ensamblado adecuado:

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

El método CSharpCompilation.AddReferences agrega referencias a la compilación. El método MetadataReference.CreateFromFile carga un ensamblado como referencia.

Consultar el modelo semántico

Una vez que tenga una Compilation, puede pedirle un SemanticModel para cualquier SyntaxTree incluido en esa Compilation. Puede considerar el modelo semántico como el origen de toda la información que normalmente obtendría de IntelliSense. Un SemanticModel puede responder a preguntas como "¿Qué nombres entran en el ámbito de esta ubicación?", "¿A qué miembros se puede acceder desde este método?", "¿Qué variables se usan en este bloque de texto?" y "¿A qué hace referencia este nombre o expresión?". Agregue esta instrucción para crear el modelo semántico:

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Enlazar un nombre

La Compilation crea el SemanticModel desde el SyntaxTree. Después de crear el modelo, puede consultarlo para buscar la primera directiva `using` y recuperar la información de símbolo del espacio de nombres `System`. Agregue estas dos líneas en su método `Main` para crear el modelo semántico y recuperar el símbolo de la primera instrucción using:

```
// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

En el código anterior se muestra cómo enlazar el nombre de la primera directiva de `using` para recuperar un Microsoft.CodeAnalysis.SymbolInfo para el espacio de nombres `System`. El código anterior también muestra que usa el **modelo de sintaxis** para buscar la estructura del código y usa el **modelo semántico** para entender su significado. El **modelo de sintaxis** busca la cadena `System` en la instrucción using. El **modelo semántico** tiene toda la información sobre los tipos definidos en el espacio de nombres `System`.

Desde el objeto SymbolInfo puede obtener el Microsoft.CodeAnalysis.ISymbol mediante la propiedad SymbolInfo.Symbol. Esta propiedad devuelve el símbolo al que hace referencia esta expresión. Para las expresiones

que no hacen referencia a ningún elemento (por ejemplo, los literales numéricos), esta propiedad es `null`. Cuando el SymbolInfo.Symbol no es NULL, el ISymbol.Kind denota el tipo del símbolo. En este ejemplo, la propiedad ISymbol.Kind es un SymbolKind.Namespace. Agregue el código siguiente al método `Main`. Recupera el símbolo del espacio de nombres `System` y, después, muestra todos los espacios de nombres secundarios que se declaran en el espacio de nombres `System`:

```
var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())
{
    Console.WriteLine(ns);
}
```

Ejecute el programa y debería ver la siguiente salida:

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

> **NOTE**
> La salida no incluye todos los espacios de nombres que son secundarios del espacio de nombres `System`. Muestra cada espacio de nombres que se encuentra en esta compilación, que solo hace referencia al ensamblado donde se declara `System.String`. Esta compilación no conoce ningún espacio de nombres declarado en otros ensamblados.

Enlazar una expresión

El código anterior muestra cómo buscar un símbolo al enlazarlo a un nombre. Hay otras expresiones en un programa de C# que se pueden enlazar que no son nombres. Para demostrar esta funcionalidad, accederemos al enlace a un literal de cadena sencillo.

El programa "Hola mundo" contiene una Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax, la cadena "Hello, World!" que se muestra en la consola.

Encontrará la cadena "Hello, World!" si busca el literal de cadena único en el programa. A continuación, una vez que haya encontrado el nodo de sintaxis, obtenga la información de tipo de ese nodo del modelo semántico. Agregue el código siguiente al método `Main`:

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
    .OfType<LiteralExpressionSyntax>()
    .Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

La estructura Microsoft.CodeAnalysis.TypeInfo incluye una propiedad TypeInfo.Type que permite el acceso a la

información semántica sobre el tipo del literal. En este ejemplo, es el tipo `string`. Agregue una declaración que asigne esta propiedad a una variable local:

```
var stringTypeSymbol = (INamedTypeSymbol)literalInfo.Type;
```

Para finalizar este tutorial, crearemos una consulta LINQ que crea una secuencia de todos los métodos públicos declarados en el tipo `string` que devuelven una `string`. Esta consulta es más compleja, así que la compilaremos línea a línea y, después, la volveremos a construir como una única consulta. El origen de esta consulta es la secuencia de todos los miembros declarados en el tipo `string`:

```
var allMembers = stringTypeSymbol.GetMembers();
```

Esa secuencia de origen contiene todos los miembros, incluidas las propiedades y los campos, de modo que tiene que filtrarlos con el método ImmutableArray<T>.OfType para buscar los elementos que son objetos Microsoft.CodeAnalysis.IMethodSymbol:

```
var methods = allMembers.OfType<IMethodSymbol>();
```

A continuación, agregue otro filtro para devolver solo aquellos métodos que son públicos y devuelven una `string`:

```
var publicStringReturningMethods = methods
    .Where(m => m.ReturnType.Equals(stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Seleccione solo la propiedad name y solo los nombres distintos; para ello, elimine las sobrecargas:

```
var distinctMethods = publicStringReturningMethods.Select(m => m.Name).Distinct();
```

También puede compilar la consulta completa con la sintaxis de consulta LINQ y, después, mostrar todos los nombres de método en la consola:

```
foreach (string name in (from method in stringTypeSymbol
                         .GetMembers().OfType<IMethodSymbol>()
                         where method.ReturnType.Equals(stringTypeSymbol) &&
                         method.DeclaredAccessibility == Accessibility.Public
                         select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Compile y ejecute el programa. Debería ver los siguientes resultados:

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

Ha usado la API semántica para buscar y mostrar información sobre los símbolos que forman parte de este programa.

Introducción a la transformación de sintaxis

29/06/2020 • 24 minutes to read • Edit Online

Este tutorial se basa en conceptos y técnicas explorados en los tutoriales rápidos Introducción al análisis de sintaxis e Introducción al análisis semántico. Si aún no lo ha hecho, debería completar esos tutoriales antes de comenzar con este.

En este tutorial rápido, se exploran las técnicas para crear y transformar árboles de sintaxis. En combinación con las técnicas que aprendió en los tutoriales anteriores, podrá crear la primera refactorización de línea de comandos.

Instrucciones de instalación: Instalador de Visual Studio

Hay dos maneras distintas de buscar el **SDK de .NET Compiler Platform** en el **Instalador de Visual Studio**:

Instalación con el Instalador de Visual Studio: visualización de cargas de trabajo

El SDK de .NET Compiler Platform no se selecciona automáticamente como parte de la carga de trabajo de desarrollo de extensiones de Visual Studio. Se debe seleccionar como un componente opcional.

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Active la carga de trabajo **Desarrollo de extensiones de Visual Studio**.
4. Abra el nodo **Desarrollo de extensiones de Visual Studio** en el árbol de resumen.
5. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en última posición bajo los componentes opcionales.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Abra el nodo **Componentes individuales** en el árbol de resumen.
2. Active la casilla **Editor de DGML**.

Instalación con el Instalador de Visual Studio: pestaña Componentes individuales

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Haga clic en la pestaña **Componentes individuales**.
4. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en la parte superior bajo la sección **Compiladores, herramientas de compilación y tiempos de ejecución**.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Active la casilla **Editor de DGML**. La encontrará en la sección **Herramientas de código**.

Inmutabilidad y .NET Compiler Platform

La **inmutabilidad** es un principio fundamental de .NET Compiler Platform. Las estructuras de datos inmutables no se pueden cambiar una vez creadas. Las estructuras de datos inmutables se pueden compartir y someter al análisis de varios consumidores al mismo tiempo sin riesgo alguno. No se corre el peligro de que la acción de un consumidor afecte a otro de manera impredecible. El analizador no necesita bloqueos u otras medidas de simultaneidad. Esta regla se aplica a los árboles de sintaxis, las compilaciones, los símbolos, los modelos semánticos y cualquier otra estructura de datos que encuentre. En lugar de modificar las estructuras existentes, las API crean nuevos objetos según las diferencias especificadas en los antiguos. Este concepto se aplica a los árboles de sintaxis para crear nuevos árboles que usan transformaciones.

Creación y transformación de árboles

Elija una de las dos estrategias para las transformaciones de sintaxis. Los patrones de diseño **Factory Method** son útiles cuando se buscan nodos específicos para reemplazar, o ubicaciones específicas donde desea insertar el nuevo código. Las **reescrituras** son la mejor opción si desea buscar patrones de código que se van a reemplazar en todo un proyecto.

Creación de nodos con patrones de diseño Factory Method

En la primera transformación de sintaxis se muestran los patrones de diseño Factory Method. Va a reemplazar una instrucción `using System.Collections;` por una instrucción `using System.Collections.Generic;`. En este ejemplo se muestra cómo crear objetos Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode mediante los patrones de diseño Factory Method Microsoft.CodeAnalysis.CSharp.SyntaxFactory. Para cada tipo de **nodo**, **token** o **curiosidades** hay un patrón Factory Method que crea una instancia de ese tipo. Creará árboles de sintaxis mediante la composición de nodos jerárquicamente de abajo arriba. Luego, transformará el programa existente para reemplazar los nodos existentes por el nuevo árbol que ha creado.

Inicie Visual Studio y cree un proyecto de **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente) de C#. En Visual Studio, elija **Archivo** > **Nuevo** > **Proyecto** para mostrar el cuadro de diálogo Nuevo proyecto. En **Visual C#** > **Extensibilidad**, elija **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente). Este tutorial rápido tiene dos proyectos de ejemplo, así que llame a la solución **SyntaxTransformationQuickStart** y al proyecto **ConstructionCS**. Haga clic en **Aceptar**.

Este proyecto usa los métodos de clase Microsoft.CodeAnalysis.CSharp.SyntaxFactory para construir un Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax que representa el espacio de nombres `System.Collections.Generic`.

Agregue esta directiva a la parte superior de `Program.cs`.

```
using  Microsoft.CodeAnalysis.CSharp;
using  System;
```

Creará **nodos de sintaxis de nombre** para generar el árbol que representa la instrucción `using System.Collections.Generic;`. NameSyntax es la clase base para los cuatro tipos de nombres que aparecen en C#. Junte estos cuatro tipos de nombres para crear cualquier nombre que pueda aparecer en el lenguaje C#:

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax, que representa los nombres de identificador único simple, como `System` y `Microsoft`.
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax, que representa un nombre de tipo o método genérico, como `List<int>`.
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax, que representa un nombre completo del formulario `<left-name>.<right-identifier-or-generic-name>`, como `System.IO`.
- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax, que representa un nombre que usa un alias de ensamblado externo, como `LibraryV2::Foo`.

Use el método IdentifierName(String) para crear un nodo NameSyntax. Agregue el código siguiente al método `Main` en `Program.cs`:

```
NameSyntax name = SyntaxFactory.IdentifierName("System");
Console.WriteLine($"\tCreated the identifier {name.ToString()}");
```

El código anterior crea un objeto IdentifierNameSyntax y lo asigna a la variable `name`. Muchas de las API de Roslyn devuelven clases base para que sea más fácil trabajar con tipos relacionados. La variable `name`, NameSyntax, puede volver a usarse al crear QualifiedNameSyntax. No use la inferencia de tipo al crear el ejemplo. Podrá automatizar

ese paso en este proyecto.

Ha creado el nombre. Ahora, es el momento de crear más nodos en el árbol mediante la creación de QualifiedNameSyntax. El nuevo árbol usa `name` como parte izquierda del nombre, y un nuevo IdentifierNameSyntax para el espacio de nombres `Collections` como parte derecha de QualifiedNameSyntax. Agrega el código siguiente a `program.cs`:

```
name = QualifiedName(name, SyntaxFactory.IdentifierName("Collections"));
Console.WriteLine(name.ToString());
```

Vuelva a ejecutar el código y observe los resultados. Está creando un árbol de nodos que representa código. Continuará con este patrón para compilar QualifiedNameSyntax para el espacio de nombres `System.Collections.Generic`. Agrega el código siguiente a `Program.cs`:

```
name = QualifiedName(name, SyntaxFactory.IdentifierName("Generic"));
Console.WriteLine(name.ToString());
```

Ejecute el programa de nuevo para ver si ha creado el árbol para agregar el código.

Creación de un árbol modificado

Ha creado un pequeño árbol de sintaxis que contiene una instrucción. Las API para crear nuevos nodos son la opción correcta para crear instrucciones únicas u otros bloques de código pequeño. Sin embargo, para generar bloques más grandes de código, debe usar los métodos que reemplazan nodos o insertan nodos en un árbol existente. Recuerde que los árboles de sintaxis son inmutables. La **API de sintaxis** no proporciona ningún mecanismo para modificar un árbol de sintaxis existente después de la construcción. En su lugar, proporciona métodos que generan nuevos árboles en función de los cambios en los existentes. Se han definido métodos `With*` en clases concretas que se derivan de SyntaxNode o en métodos de extensión declarados en la clase SyntaxNodeExtensions. Estos métodos crean un nuevo nodo al aplicar cambios a las propiedades del elemento secundario de un nodo existente. Además, el método de extensión ReplaceNode se puede utilizar para reemplazar un nodo descendiente en un subárbol. Este método también actualiza el elemento primario para que apunte al formulario secundario recién creado y repite este proceso por todo el árbol: un proceso conocido como *volver a hacer girar el árbol*.

El siguiente paso consiste en crear un árbol que representa todo un programa (pequeño) y, a continuación, modificarlo. Agregue el siguiente código al principio de la clase `Program`:

```
        private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

> **NOTE**
> El código de ejemplo utiliza el espacio de nombres `System.Collections` y no el espacio de nombres `System.Collections.Generic`.

A continuación, agregue el código siguiente a la parte inferior del método `Main` para analizar el texto y crear un árbol:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

En este ejemplo se utiliza el método WithName(NameSyntax) para reemplazar el nombre de un nodo UsingDirectiveSyntax por el que se creó en el código anterior.

Cree un nuevo nodo UsingDirectiveSyntax mediante el método WithName(NameSyntax) para actualizar el nombre `System.Collections` con el nombre que creó en el código anterior. Agregue el siguiente código en la parte inferior del método `Main`:

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
Console.WriteLine(root.ToString());
```

Ejecute el programa y observe con detenimiento el resultado. `newusing` todavía no se ha colocado en el árbol de la raíz. No se ha modificado el árbol original.

Agregue el siguiente código utilizando el método de extensión ReplaceNode para crear un nuevo árbol. El nuevo árbol es el resultado de reemplazar la importación existente por el nodo `newUsing` actualizado. Asigne este nuevo árbol a la variable `root` existente:

```
root = root.ReplaceNode(oldUsing, newUsing);
Console.WriteLine(root.ToString());
```

Ejecute el programa otra vez. Esta vez el árbol importa correctamente el espacio de nombres `System.Collections.Generic`.

Transformación de árboles mediante `SyntaxRewriters`

Los métodos `With*` y ReplaceNode proporcionan un medio cómodo para transformar ramas individuales de un árbol de sintaxis. La clase Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter realiza varias transformaciones en un árbol de sintaxis. La clase Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter es una subclase de Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>. CSharpSyntaxRewriter aplica una transformación a un tipo específico de SyntaxNode. Puede aplicar transformaciones a varios tipos de objetos SyntaxNode dondequiera que aparezcan en un árbol de sintaxis. En el segundo proyecto de este tutorial rápido se crea una refactorización de línea de comandos que quita tipos explícitos en declaraciones de variable local en cualquier lugar en que pudiera utilizarse una inferencia de tipos.

Cree un proyecto de **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente) de C#. En Visual Studio, haga clic en el nodo de la solución `SyntaxTransformationQuickStart`. Elija **Agregar** > **Nuevo proyecto** para mostrar el **cuadro de diálogo Nuevo proyecto**. En **Visual C#** > **Extensibilidad**, elija **Stand-Alone Code Analysis Tool** (Herramienta de análisis de código independiente). Proporcione un nombre al proyecto `TransformationCS` y haga clic en Aceptar.

El primer paso es crear una clase que se derive de CSharpSyntaxRewriter para realizar las transformaciones. Agregue un nuevo archivo de clase al proyecto. En Visual Studio, elija **Proyecto** > **Agregar clase...** . En el cuadro

de diálogo **Agregar nuevo elemento**, escriba `TypeInferenceRewriter.cs` como nombre de archivo.

Agregue las siguientes directivas using al archivo `TypeInferenceRewriter.cs`:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
using Microsoft.CodeAnalysis.CSharp;
```

A continuación, haga que la clase `TypeInferenceRewriter` se extienda a la clase CSharpSyntaxRewriter:

```
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Agregue el código siguiente para declarar un campo privado de solo lectura para que contenga un SemanticModel e inicializarlo en el constructor. Necesitará este campo más adelante para determinar donde se puede usar la inferencia de tipos:

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

Invalide el método VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax):

```
public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{

}
```

> **NOTE**
> Muchas de las API de Roslyn declaran tipos de valor devuelto que son clases base de los tipos en tiempo de ejecución reales devueltos. En muchos escenarios, un tipo de nodo puede reemplazarse por otro tipo de nodo por completo, e incluso eliminarse. En este ejemplo, el método VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) devuelve un SyntaxNode, en lugar del tipo derivado de LocalDeclarationStatementSyntax. Este sistema de reescritura devuelve un nuevo nodo LocalDeclarationStatementSyntax basado en el existente.

Este tutorial rápido controla las declaraciones de variable locales. Puede ampliarlo para otras declaraciones como bucles `foreach`, bucles `for`, expresiones LINQ y expresiones lambda. Además, este sistema de reescritura transformará solo las declaraciones de la forma más sencilla:

```
Type variable = expression;
```

Si desea explorar por su cuenta, considere la posibilidad de extender el ejemplo finalizado para estos tipos de declaraciones de variable:

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
     variable2 = expression2;
// No initializer.
Type variable;
```

Agregue el código siguiente al cuerpo del método `VisitLocalDeclarationStatement` para que omita la reescritura de

estas formas de declaraciones:

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

El método indica que no se realiza la reescritura mediante la devolución del parámetro `node` sin modificar. Si ninguna de esas expresiones `if` son verdaderas, el nodo representa una declaración posible con la inicialización. Agregue estas instrucciones para extraer el nombre del tipo especificado en la declaración y asócielo con el campo SemanticModel para obtener un símbolo de tipo:

```
VariableDeclaratorSyntax declarator = node.Declaration.Variables.First();
TypeSyntax variableTypeName = node.Declaration.Type;

ITypeSymbol variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Ahora, agregue esta instrucción para enlazar la expresión de inicializador:

```
TypeInfo initializerInfo = SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Por último, agregue la siguiente instrucción `if` para reemplazar el nombre de tipo existente por la palabra clave `var` si el tipo de la expresión de inicializador coincide con el tipo especificado:

```
if (variableType == initializerInfo.Type)
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

El atributo condicional es necesario porque la declaración puede convertir la expresión de inicializador en una interfaz o clase base. Si se desea, los tipos del lado izquierdo y derecho de la asignación no coinciden. El hecho de quitar el tipo explícito en estos casos también cambiaría la semántica de un programa. `var` se especifica como un identificador en lugar de una palabra clave porque `var` es una palabra clave contextual. Las curiosidades iniciales y finales (espacios en blanco) se transfieren desde el nombre de tipo anterior a la palabra clave `var` para mantener el espacio en blanco vertical y la sangría. Es más fácil utilizar `ReplaceNode` en lugar de `With*` para transformar el LocalDeclarationStatementSyntax porque el nombre de tipo es realmente el descendiente del elemento secundario de la instrucción de declaración.

Ha terminado el `TypeInferenceRewriter`. Ahora vuelva a su archivo `Program.cs` para finalizar el ejemplo. Cree una prueba Compilation y obtenga SemanticModel de ella. Use ese SemanticModel para probar su `TypeInferenceRewriter`. Llevará a cabo este último paso. Mientras tanto, declare una variable de marcador de posición que represente la compilación de prueba:

```
Compilation test = CreateTestCompilation();
```

Después de un momento de pausa, debería aparecer un subrayado ondulado de error que indica que no existe ningún método `CreateTestCompilation`. Presione **CTRL+Punto** para abrir la bombilla y, a continuación, presione Entrar para invocar el comando **Generar código auxiliar del método**. Este comando generará un código auxiliar para el método `CreateTestCompilation` en la clase `Program`. Podrá volver a rellenar este método más adelante:

Escriba el código siguiente para recorrer en iteración cada SyntaxTree en la prueba Compilation. Para cada una de ellas, inicialice un nuevo `TypeInferenceRewriter` con el SemanticModel para ese árbol:

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

Dentro de la instrucción `foreach` que ha creado, agregue el código siguiente para realizar la transformación en cada árbol de origen. Este código escribe condicionalmente el nuevo árbol transformado si se hicieron modificaciones. El sistema de reescritura sólo debe modificar un árbol si encuentra una o más declaraciones de variables locales que pudieron simplificarse mediante la inferencia de tipos:

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

Debería ver los subrayados ondulados bajo el código `File.WriteAllText`. Seleccione la bombilla y agregue la

instrucción `using System.IO;` necesaria.

Casi ha terminado. Queda un último paso: crear una prueba Compilation. Puesto que no ha utilizado ninguna inferencia de tipos durante este tutorial rápido, habría sido un caso de prueba perfecto. Desafortunadamente, la creación de una compilación desde un archivo de proyecto de C# queda fuera del ámbito de este tutorial. Pero afortunadamente, si ha seguido las instrucciones con cuidado, hay esperanza. Reemplace el contenido del método `CreateTestCompilation` con el código siguiente. Crea una compilación de prueba que casualmente coincide con el proyecto descrito en este tutorial rápido:

```
String programPath = @"..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
               CSharpSyntaxTree.ParseText(programText)
                                .WithFilePath(programPath);

String rewriterPath = @".\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
               CSharpSyntaxTree.ParseText(rewriterText)
                                .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
        MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =
        MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location);

MetadataReference[] references = { mscorlib, codeAnalysis, csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

Cruce los dedos y ejecute el proyecto. En Visual Studio, elija **Depurar** > **Iniciar depuración**. Visual Studio le debería notificar que los archivos de su proyecto han cambiado. Haga clic en "**Sí a todo**" para volver a cargar los archivos modificados. Examínelos para observar su genialidad. Observe que el código se ve mucho más limpio sin todos los especificadores de tipo explícitos y redundantes.

¡Enhorabuena! Ha usado las **API de compilador** para escribir su propia refactorización que busca en todos los archivos de un proyecto C# ciertos patrones sintácticos, analiza la semántica del código fuente que coincide con esos patrones y la transforma. ¡Ya es oficialmente un autor de refactorización!

Tutorial: Crear el primer analizador y la corrección de código

29/06/2020 • 46 minutes to read • Edit Online

El SDK de .NET Compiler Platform proporciona las herramientas necesarias para crear advertencias personalizadas destinadas al código de C# o de Visual Basic. Su **analizador** contiene código que reconoce las infracciones de la regla. La **corrección del código** contiene el código que corrige la infracción. Las reglas implementadas pueden ser cualquier elemento de la estructura de código para codificar el estilo de las convenciones de nomenclatura y mucho más. .NET Compiler Platform proporciona el marco para ejecutar el análisis a medida que los desarrolladores escriben código y todas las características de la interfaz de usuario de Visual Studio para corregir código: mostrar líneas de subrayado en el editor, rellenar la lista de errores de Visual Studio, crear las sugerencias con "bombillas" y mostrar la vista previa enriquecida de las correcciones sugeridas.

En este tutorial, explorará la creación de un **analizador** y una **corrección de código** complementaria con el uso de las API de Roslyn. Un analizador es una manera de realizar análisis de código fuente y notificar un problema al usuario. Opcionalmente, un analizador también puede proporcionar una corrección de código que representa una modificación del código fuente del usuario. Este tutorial crea un analizador que busca declaraciones de variable local que podrían declararse mediante el modificador `const`, aunque no están. La corrección de código complementaria modifica esas declaraciones para agregar el modificador `const`.

Requisitos previos

- Visual Studio 2017
- Visual Studio 2019

Debe instalar el **SDK de .NET Compiler Platform** a través del Instalador de Visual Studio:

Instrucciones de instalación: Instalador de Visual Studio

Hay dos maneras distintas de buscar el **SDK de .NET Compiler Platform** en el **Instalador de Visual Studio**:

Instalación con el Instalador de Visual Studio: visualización de cargas de trabajo

El SDK de .NET Compiler Platform no se selecciona automáticamente como parte de la carga de trabajo de desarrollo de extensiones de Visual Studio. Se debe seleccionar como un componente opcional.

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.
3. Active la carga de trabajo **Desarrollo de extensiones de Visual Studio**.
4. Abra el nodo **Desarrollo de extensiones de Visual Studio** en el árbol de resumen.
5. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en última posición bajo los componentes opcionales.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Abra el nodo **Componentes individuales** en el árbol de resumen.
2. Active la casilla **Editor de DGML**.

Instalación con el Instalador de Visual Studio: pestaña Componentes individuales

1. Ejecute el **Instalador de Visual Studio**.
2. Haga clic en **Modificar**.

3. Haga clic en la pestaña **Componentes individuales**.
4. Active la casilla **SDK de .NET Compiler Platform**. La encontrará en la parte superior bajo la sección **Compiladores, herramientas de compilación y tiempos de ejecución**.

Opcionalmente, también le interesará que el **Editor de DGML** muestre los gráficos en el visualizador:

1. Active la casilla **Editor de DGML**. La encontrará en la sección **Herramientas de código**.

Hay varios pasos para crear y validar su analizador:

1. Crear la solución.
2. Registrar el nombre del analizador y la descripción.
3. Notificar las recomendaciones y las advertencias del analizador.
4. Implementar la corrección de código para aceptar las recomendaciones.
5. Mejorar el análisis mediante las pruebas unitarias.

Exploración de la plantilla del analizador

El analizador notifica al usuario las declaraciones de variable local que se pueden convertir en constantes locales. Por ejemplo, considere el siguiente código:

```
int x = 0;
Console.WriteLine(x);
```

En el código anterior, a `x` se le asigna un valor constante y nunca se modifica. Se puede declarar con el modificador `const`:

```
const int x = 0;
Console.WriteLine(x);
```

El análisis para determinar si una variable se puede convertir en constante, que requiere un análisis sintáctico, un análisis constante de la expresión del inicializador y un análisis del flujo de datos para garantizar que no se escriba nunca en la variable. .NET Compiler Platform proporciona las API que facilita la realización de este análisis. El primer paso es crear un proyecto en C# del **analizador con corrección de código**.

- En Visual Studio, elija **Archivo > Nuevo > Proyecto...** para mostrar el cuadro de diálogo Nuevo proyecto.
- En **Visual C# > Extensibilidad**, elija **Analizador con corrección de código (.NET Standard)** .
- Asigne al proyecto el nombre "**MakeConst**" y haga clic en Aceptar.

La plantilla del analizador con corrección de código crea tres proyectos: uno contiene el analizador y la corrección de código, el segundo es un proyecto de prueba unitaria y el tercero es el proyecto de VSIX. El proyecto de inicio predeterminado es el proyecto de VSIX. Presione **F5** para iniciar el proyecto de VSIX. De esta forma se inicia una segunda instancia de Visual Studio que ha cargado el nuevo analizador.

> **TIP**
> Al ejecutar el analizador, inicie una segunda copia de Visual Studio. Esta segunda copia usa un subárbol del registro diferente para almacenar la configuración. Le permite diferenciar la configuración visual en las dos copias de Visual Studio. Puede elegir un tema diferente para la ejecución experimental de Visual Studio. Además, no mueva la configuración o el inicio de sesión a la cuenta de Visual Studio con la ejecución experimental de Visual Studio. Así se mantiene una configuración diferente.

En la segunda instancia de Visual Studio que acaba de iniciar, cree un proyecto de aplicación de consola de C# (los proyectos de .NET Core o .NET Framework funcionarán; los analizadores funcionan a nivel de origen). Mantenga el

mouse sobre el token con un subrayado ondulado y aparecerá el texto de advertencia proporcionado por un analizador.

La plantilla crea un analizador que notifica una advertencia en cada declaración de tipo, donde el nombre de tipo contiene letras minúsculas, tal como se muestra en la ilustración siguiente:

```
namespace ConsoleApp1
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

La plantilla también proporciona una corrección de código que cambia cualquier nombre de tipo que contiene caracteres en minúsculas a mayúsculas. Puede hacer clic en la bombilla mostrada con la advertencia para ver los cambios sugeridos. Al aceptar los cambios sugeridos, se actualizan el nombre de tipo y todas las referencias a dicho tipo en la solución. Ahora que ha visto el analizador inicial en acción, cierre la segunda instancia de Visual Studio y vuelva a su proyecto de analizador.

No tiene que iniciar una segunda copia de Visual Studio, y cree código para probar todos los cambios en el analizador. La plantilla también crea un proyecto de prueba unitaria de forma automática. Este proyecto contiene dos pruebas. `TestMethod1` muestra el formato típico de una prueba que analiza el código sin que se desencadene un diagnóstico. `TestMethod2` muestra el formato de una prueba que desencadena un diagnóstico y, a continuación, se aplica una corrección de código sugerida. Al crear el analizador y la corrección de código, deberá escribir pruebas para diferentes estructuras de código para verificar el trabajo. Las pruebas unitarias de los analizadores son mucho más rápidas que las pruebas interactivas con Visual Studio.

> **TIP**
> Las pruebas unitarias del analizador son una herramienta magnífica si se sabe qué construcciones de código deben y no deben desencadenar el analizador. Cargar el analizador en otra copia de Visual Studio es una herramienta magnífica para explorar y buscar construcciones en las que puede no haber pensado todavía.

Creación de registros del analizador

La plantilla crea la clase `DiagnosticAnalyzer` inicial en el archivo **MakeConstAnalyzer.cs**. Este analizador inicial muestra dos propiedades importantes de cada analizador.

- Cada analizador de diagnóstico debe proporcionar un atributo `[DiagnosticAnalyzer]` que describe el lenguaje en el que opera.
- Cada analizador de diagnóstico debe derivar de la clase DiagnosticAnalyzer.

La plantilla también muestra las características básicas que forman parte de cualquier analizador:

1. Registre acciones. Las acciones representan los cambios de código que deben desencadenar el analizador para examinar el código para las infracciones. Cuando Visual Studio detecta las modificaciones del código que coinciden con una acción registrada, llama al método registrado del analizador.
2. Cree diagnósticos. Cuando el analizador detecta una infracción, crea un objeto de diagnóstico que Visual Studio usa para notificar la infracción al usuario.

Registre acciones en la invalidación del método DiagnosticAnalyzer.Initialize(AnalysisContext). En este tutorial, repasará **nodos de sintaxis** que buscan declaraciones locales y verá cuáles de ellos tienen valores constantes. Si

una declaración puede ser constante, el analizador creará y notificará un diagnóstico.

El primer paso es actualizar las constantes de registro y el método `Initialize`, por lo que estas constantes indican su analizador "Make Const". La mayoría de las constantes de cadena se definen en el archivo de recursos de cadena. Debe seguir dicha práctica para una localización más sencilla. Abra el archivo **Resources.resx** para el proyecto de analizador **MakeConst**. Muestra el editor de recursos. Actualice los recursos de cadena como sigue:

- Cambie `AnalyzerTitle` por "La variable puede convertirse en constante".
- Cambie `AnalyzerMessageFormat` por "Puede convertirse en constante".
- Cambie `AnalyzerDescription` por "Convertir en constante".

Cambie también el menú desplegable del **modificador de acceso** por `public`. De esta forma, se facilita el uso de estas constantes en las pruebas unitarias. Cuando haya terminado, el editor de recursos debe aparecer como se muestra en la figura siguiente:

Name	Value	Comment
AnalyzerTitle	Variable can be made constant.	The title of the diagnostic.
AnalyzerMessageFormat	can be made constant	The format-able message the diagnostic displays.
AnalyzerDescription	Make Constant.	An optional longer localizable description of the diagnostic.

Los cambios restantes están en el archivo del analizador. Abra **MakeConstAnalyzer.cs** en Visual Studio. Cambie la acción registrada de una que actúa en los símbolos a una que actúa en la sintaxis. En el método `MakeConstAnalyzerAnalyzer.Initialize`, busque la línea que registra la acción en los símbolos:

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Reemplácela por la línea siguiente:

```
context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.Analyze | GeneratedCodeAnalysisFlags.None);
context.EnableConcurrentExecution();
context.RegisterSyntaxNodeAction(AnalyzeNode, SyntaxKind.LocalDeclarationStatement);
```

Después de este cambio, puede eliminar el método `AnalyzeSymbol`. Este analizador examina SyntaxKind.LocalDeclarationStatement, no las instrucciones SymbolKind.NamedType. Tenga en cuenta que `AnalyzeNode` tiene un subrayado ondulado rojo debajo. El código recién agregado hace referencia a un método `AnalyzeNode` que no se ha declarado. Declare dicho método con el siguiente código:

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Cambie `Category` por "Uso" en **MakeConstAnalyzer.cs** como se muestra en el código siguiente:

```
private const string Category = "Usage";
```

Búsqueda de las declaraciones locales que podrían ser constantes

Es el momento de escribir la primera versión del método `AnalyzeNode`. Debe buscar una sola declaración local que podría ser `const` pero no lo es, al igual que el código siguiente:

```
int x = 0;
Console.WriteLine(x);
```

El primer paso es encontrar las declaraciones locales. Agregue el código siguiente a `AnalyzeNode` en **MakeConstAnalyzer.cs**:

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

Esta conversión siempre se realiza correctamente porque el analizador registró los cambios de las declaraciones locales, y solo las declaraciones locales. Ningún otro tipo de nodo desencadena una llamada al método `AnalyzeNode`. A continuación, compruebe la declaración de cualquier modificador `const`. Si la encuentra, devuélvala de inmediato. El código siguiente busca cualquier modificador `const` en la declaración local:

```
// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

Por último, deberá comprobar que la variable podría ser `const`. Esto significa asegurarse de que nunca se asigne después de inicializarse.

Realizará algún análisis semántico con SyntaxNodeAnalysisContext. Use el argumento `context` para determinar si la declaración de variable local puede convertirse en `const`. Una clase Microsoft.CodeAnalysis.SemanticModel representa toda la información semántica en un solo archivo de origen. Puede obtener más información en el artículo que trata los modelos semánticos. Deberá usar Microsoft.CodeAnalysis.SemanticModel para realizar análisis de flujo de datos en la instrucción de declaración local. A continuación, use los resultados de este análisis de flujo de datos para garantizar que la variable local no se escriba con un valor nuevo en cualquier otro lugar. Llame al método de extensión GetDeclaredSymbol para recuperar ILocalSymbol para la variable y compruebe si no está incluido en la colección DataFlowAnalysis.WrittenOutside del análisis de flujo de datos. Agregue el código siguiente al final del método `AnalyzeNode`:

```
// Perform data flow analysis on the local declaration.
var dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
var variable = localDeclaration.Declaration.Variables.Single();
var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

El código recién agregado garantiza que no se modifique la variable y que se pueda convertir por tanto en `const`. Es el momento de generar el diagnóstico. Agregue el código siguiente a la última línea en `AnalyzeNode`:

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation()));
```

Puede comprobar el progreso presionando **F5** para ejecutar el analizador. Puede cargar la aplicación de consola que creó anteriormente y después agregar el siguiente código de prueba:

```
int x = 0;
Console.WriteLine(x);
```

Debe aparecer la bombilla, y el analizador debe informar de un diagnóstico. Sin embargo, la bombilla todavía indica que la plantilla generó la corrección de código e indica que se puede convertir en mayúsculas. En la sección siguiente se explica cómo escribir la corrección de código.

Escritura de la corrección de código

Un analizador puede proporcionar una o varias correcciones de código. Una corrección de código define una edición que soluciona el problema notificado. Para el analizador que ha creado, puede proporcionar una corrección de código que inserta la palabra clave const:

```
const int x = 0;
Console.WriteLine(x);
```

El usuario la elige en la interfaz de usuario de la bombilla del editor, y Visual Studio cambia el código.

Abra el archivo **MakeConstCodeFixProvider.cs** agregado por la plantilla. Esta corrección de código ya está conectada con el identificador de diagnóstico generado por el analizador de diagnóstico, pero aún no implementa la transformación de código correcta. En primer lugar debe quitar parte del código de la plantilla. Cambie la cadena de título por "Convertir en constante":

```
private const string title = "Make constant";
```

Después, elimine el método `MakeUppercaseAsync`. Ya no se aplica.

Todos los proveedores de corrección de código se derivan de CodeFixProvider. Todas invalidan CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext) para notificar las correcciones de código disponibles. En `RegisterCodeFixesAsync`, cambie el tipo de nodo antecesor que está buscando por LocalDeclarationStatementSyntax para que coincida con el diagnóstico:

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>
().First();
```

A continuación, cambie la última línea para registrar una corrección de código. La corrección creará un documento que resulta de agregar el modificador `const` a una declaración existente:

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: title,
        createChangedDocument: c => MakeConstAsync(context.Document, declaration, c),
        equivalenceKey: title),
    diagnostic);
```

Observará un subrayado ondulado rojo en el código que acaba de agregar en el símbolo `MakeConstAsync`. Agregue una declaración para `MakeConstAsync` como el código siguiente:

```
private async Task<Document> MakeConstAsync(Document document,
   LocalDeclarationStatementSyntax localDeclaration,
   CancellationToken cancellationToken)
{
}
```

El nuevo método `MakeConstAsync` transformará la clase Document que representa el archivo de origen del usuario en una nueva clase Document que ahora contiene una declaración `const`.

Se crea un token de palabra clave `const` para insertarlo en la parte delantera de la instrucción de declaración. Tenga cuidado de quitar primero cualquier curiosidad inicial del primer token de la instrucción de declaración y adjúntela al token `const`. Agregue el código siguiente al método `MakeConstAsync`:

```
// Remove the leading trivia from the local declaration.
var firstToken = localDeclaration.GetFirstToken();
var leadingTrivia = firstToken.LeadingTrivia;
var trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
var constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

A continuación, agregue el token `const` a la declaración con el siguiente código:

```
// Insert the const token into the modifiers list, creating a new modifiers list.
var newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
var newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Después aplique formato a la nueva declaración para que coincida con las reglas de formato de C#. Aplicar formato a los cambios para que coincidan con el código existente mejora la experiencia. Agregue la instrucción siguiente inmediatamente después del código existente:

```
// Add an annotation to format the new local declaration.
var formattedLocal = newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

Se requiere un nuevo espacio de nombres para este código. Agregue la siguiente directiva `using` al principio del archivo:

```
using Microsoft.CodeAnalysis.Formatting;
```

El último paso es realizar la edición. Hay tres pasos para este proceso:

1. Obtenga un identificador para el documento existente.
2. Cree un documento mediante el reemplazo de la declaración existente con la nueva declaración.
3. Devuelva el nuevo documento.

Agregue el código siguiente al final del método `MakeConstAsync`:

```
// Replace the old local declaration with the new local declaration.
var oldRoot = await document.GetSyntaxRootAsync(cancellationToken);
var newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

La corrección de código está lista para probarla. Presione F5 para ejecutar el proyecto del analizador en una segunda instancia de Visual Studio. En la segunda instancia de Visual Studio, cree un proyecto de aplicación de consola de C# y agregue algunas declaraciones de variable local inicializadas con valores de constante para el método Main. Observará que se notifican como advertencias de la siguiente forma.

```
static void Main(string[] args)
{
    int i = 1;
    int j = 2;
    int k = i + j;
}
```

Ha progresado bastante. Hay subrayados ondulados debajo de las declaraciones que pueden convertirse en `const`. Pero aún queda trabajo por hacer. Esto funciona bien si agrega `const` a las declaraciones a partir de `i`, luego `j` y, por último, `k`. Sin embargo, si agrega el modificador `const` en un orden diferente, a partir de `k`, el analizador crea errores: `k` no puede declararse como `const`, a menos que `i` y `j` ya sean `const`. Tiene que realizar más análisis para asegurarse de que controla la forma en que las variables pueden declararse e inicializarse.

Creación de pruebas basadas en datos

El analizador y la corrección de código funcionan en un caso sencillo de una única declaración que puede convertirse en const. Hay varias instrucciones de declaración posibles donde esta implementación comete errores. Abordará estos casos al trabajar con la biblioteca de pruebas unitarias escrita por la plantilla. Es mucho más rápido que abrir repetidamente una segunda copia de Visual Studio.

Abra el archivo **MakeConstUnitTests.cs** en el proyecto de prueba unitaria. La plantilla creó dos pruebas que siguen los dos patrones comunes para una prueba unitaria de la corrección de código y del analizador. `TestMethod1` muestra el patrón para una prueba que garantiza que el analizador no notifique un diagnóstico cuando no debe. `TestMethod2` muestra el patrón de notificación de un diagnóstico y de ejecución de la corrección de código.

El código para casi todas las pruebas del analizador sigue uno de estos dos patrones. Para el primer paso, puede reprocesar estas pruebas como pruebas basadas en datos. Después, será fácil crear pruebas con la adición de nuevas constantes de cadena para representar diferentes entradas de prueba.

Por cuestiones de eficacia, el primer paso es refactorizar las dos pruebas en pruebas basadas en datos. Después, solo necesita definir un par de constantes de cadena para cada prueba nueva. Durante la refactorización, cambie el nombre de ambos métodos por otros nombres mejores. Reemplace `TestMethod1` con esta prueba que garantiza que no se realizará ningún diagnóstico:

```
[DataTestMethod]
[DataRow("")]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
{
    VerifyCSharpDiagnostic(testCode);
}
```

Puede crear una fila de datos para esta prueba mediante la definición de cualquier fragmento de código que no debería provocar que el diagnóstico desencadene una advertencia. Esta sobrecarga de `VerifyCSharpDiagnostic`

pasa cuando no hay ningún diagnóstico desencadenado para el fragmento de código fuente.

Después, reemplace `TestMethod2` con esta prueba que garantiza la realización de un diagnóstico y la aplicación de una corrección de código para el fragmento de código fuente:

```
[DataTestMethod]
[DataRow(LocalIntCouldBeConstant, LocalIntCouldBeConstantFixed, 10, 13)]
public void WhenDiagnosticIsRaisedFixUpdatesCode(
    string test,
    string fixTest,
    int line,
    int column)
{
    var expected = new DiagnosticResult
    {
        Id = MakeConstAnalyzer.DiagnosticId,
        Message = new LocalizableResourceString(nameof(MakeConst.Resources.AnalyzerMessageFormat),
MakeConst.Resources.ResourceManager, typeof(MakeConst.Resources)).ToString(),
        Severity = DiagnosticSeverity.Warning,
        Locations =
            new[] {
                    new DiagnosticResultLocation("Test0.cs", line, column)
                }
    };

    VerifyCSharpDiagnostic(test, expected);

    VerifyCSharpFix(test, fixTest);
}
```

El código anterior también realizó un par de cambios en el código que compila el resultado de diagnóstico esperado. Usa las constantes públicas registradas en el analizador `MakeConst`. Además, utiliza dos constantes de cadena para el origen de entrada y fijo. Agregue las siguientes constantes de cadena a la clase `UnitTest`:

```
private const string LocalIntCouldBeConstant = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            Console.WriteLine(i);
        }
    }
}";

private const string LocalIntCouldBeConstantFixed = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const int i = 0;
            Console.WriteLine(i);
        }
    }
}";
```

Ejecute estas dos pruebas para asegurarse de que pasen. En Visual Studio, abra el **Explorador de pruebas**; para ello, seleccione **Prueba** > **Windows** > **Explorador de pruebas**. Haga clic en el vínculo **Ejecutar todo**.

Creación de pruebas para declaraciones válidas

Por norma general, los analizadores deben existir lo más rápido posible, pero haciendo el mínimo trabajo. Visual Studio llama a los analizadores registrados a medida que el usuario edita el código. La capacidad de respuesta es un requisito clave. Hay varios casos de pruebas del código que no deben realizar un diagnóstico. El analizador ya controla una de esas pruebas, el caso en que una variable se asigna después de inicializarse. Agregue la siguiente constante de cadena a las pruebas para representar dicho caso:

```
private const string VariableAssigned = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            Console.WriteLine(i++);
        }
    }
}";
```

Después, agregue una fila de datos para esta prueba como se muestra en el fragmento de código siguiente:

```
[DataTestMethod]
[DataRow(""),
 DataRow(VariableAssigned)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
```

Esta prueba también pasa. Después, agregue constantes para las condiciones que aún no ha controlado:

- Declaraciones que ya son `const`, porque ya son constantes:

  ```
  private const string AlreadyConst = @"
  using System;

  namespace MakeConstTest
  {
      class Program
      {
          static void Main(string[] args)
          {
              const int i = 0;
              Console.WriteLine(i);
          }
      }
  }";
  ```

- Declaraciones que no tienen inicializador, porque no hay ningún valor para usar:

```
private const string NoInitializer = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;
            i = 0;
            Console.WriteLine(i);
        }
    }
}";
```

- Declaraciones donde el inicializador no es una constante, porque no pueden ser constantes en tiempo de compilación:

```
private const string InitializerNotConstant = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = DateTime.Now.DayOfYear;
            Console.WriteLine(i);
        }
    }
}";
```

Puede ser incluso más complicado, porque C# admite varias declaraciones como una instrucción. Considere la siguiente constante de cadena de caso de prueba:

```
private const string MultipleInitializers = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0, j = DateTime.Now.DayOfYear;
            Console.WriteLine(i, j);
        }
    }
}";
```

La variable `i` puede convertirse en constante, pero la variable `j` no puede. Por tanto, esta instrucción no puede convertirse en una declaración de constante. Agregue las declaraciones `DataRow` para todas estas pruebas:

```
[DataTestMethod]
[DataRow(""),
    DataRow(VariableAssigned),
    DataRow(AlreadyConst),
    DataRow(NoInitializer),
    DataRow(InitializerNotConstant),
    DataRow(MultipleInitializers)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
```

Vuelva a ejecutar las pruebas y, después, observará que estos nuevos casos de prueba generarán errores.

Actualización del analizador para ignorar declaraciones correctas

Necesita algunas mejoras en el método `AnalyzeNode` del analizador para filtrar el código que cumple estas condiciones. Son todas condiciones relacionadas, por lo que los cambios similares corregirán todas estas condiciones. Realice los siguientes cambios en `AnalyzeNode`:

- El análisis semántico analizó una única declaración de variable. Este código necesita estar en un bucle `foreach` que examina todas las variables declaradas en la misma instrucción.
- Cada variable declarada necesita tener un inicializador.
- El inicializador de cada variable declarada debe ser una constante de tiempo de compilación.

En el método `AnalyzeNode`, reemplace el análisis semántico original:

```
// Perform data flow analysis on the local declaration.
var dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
var variable = localDeclaration.Declaration.Variables.Single();
var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

por el siguiente fragmento de código:

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (var variable in localDeclaration.Declaration.Variables)
{
    var initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    var constantValue = context.SemanticModel.GetConstantValue(initializer.Value);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
var dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (var variable in localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis region.
    var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

El primer bucle `foreach` examina cada declaración de variable con análisis sintácticos La primera comprobación garantiza que la variable tiene un inicializador. La segunda comprobación garantiza que el inicializador es una constante. El segundo bucle tiene el análisis semántico original. Las comprobaciones semánticas se encuentran en un bucle independiente porque afectan más al rendimiento. Vuelva a ejecutar las pruebas y observará que todas pasan.

Adición de un retoque final

Casi ha terminado. Hay algunas condiciones más que el analizador tiene que cumplir. Visual Studio llama a los analizadores mientras el usuario escribe el código. Suele darse el caso de que se llama al analizador para código que no compila. El método `AnalyzeNode` del analizador de diagnóstico no comprueba si el valor de constante se puede convertir al tipo de variable. Por tanto, la implementación actual convertirá correctamente una declaración incorrecta, como int i = "abc"', en una constante local. Agregue una constante de cadena de origen para esa condición:

```
private const string DeclarationIsInvalid = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = ""abc"";
        }
    }
}";
```

Además, los tipos de referencia no se controlan correctamente. El único valor de constante permitido para un tipo

de referencia es `null`, excepto en este caso de System.String, que admite los literales de cadena. En otras palabras, `const string s = "abc"` es legal, pero `const object s = "abc"` no lo es. Este fragmento de código comprueba esa condición:

```
private const string ReferenceTypeIsntString = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            object s = ""abc"";
        }
    }
}";
```

Para ser exhaustivo, debe agregar otra prueba para asegurarse de que puede crear una declaración de constante para una cadena. El fragmento de código siguiente define el código que genera el diagnóstico y el código después de haber aplicado la corrección:

```
private const string ConstantIsString = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = ""abc"";
        }
    }
}";

private const string ConstantIsStringFixed = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const string s = ""abc"";
        }
    }
}";
```

Por último, si una variable se declara con la palabra clave `var`, la corrección de código hace una función incorrecta y genera una declaración `const var`, que el lenguaje C# no admite. Para corregir este error, la corrección de código debe reemplazar la palabra clave `var` por el nombre del tipo deducido:

```
private const string DeclarationUsesVar = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            var item = 4;
        }
    }
}";

private const string DeclarationUsesVarFixedHasType = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const int item = 4;
        }
    }
}";
private const string StringDeclarationUsesVar = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            var item = ""abc"";
        }
    }
}";
private const string StringDeclarationUsesVarFixedHasType = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const string item = ""abc"";
        }
    }
}";
```

Estos cambios actualizan las declaraciones de la fila de datos en ambas pruebas. El código siguiente muestra estas pruebas con todos los atributos de la fila de datos:

```
//No diagnostics expected to show up
[DataTestMethod]
[DataRow(""),
 DataRow(VariableAssigned),
 DataRow(AlreadyConst),
 DataRow(NoInitializer),
 DataRow(InitializerNotConstant),
 DataRow(MultipleInitializers),
 DataRow(DeclarationIsInvalid),
 DataRow(ReferenceTypeIsntString)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
{
    VerifyCSharpDiagnostic(testCode);
}

[DataTestMethod]
[DataRow(LocalIntCouldBeConstant, LocalIntCouldBeConstantFixed, 10, 13),
 DataRow(ConstantIsString, ConstantIsStringFixed, 10, 13),
 DataRow(DeclarationUsesVar, DeclarationUsesVarFixedHasType, 10, 13),
 DataRow(StringDeclarationUsesVar, StringDeclarationUsesVarFixedHasType, 10, 13)]
public void WhenDiagosticIsRaisedFixUpdatesCode(
    string test,
    string fixTest,
    int line,
    int column)
{
    var expected = new DiagnosticResult
    {
        Id = MakeConstAnalyzer.DiagnosticId,
        Message = new LocalizableResourceString(nameof(MakeConst.Resources.AnalyzerMessageFormat),
MakeConst.Resources.ResourceManager, typeof(MakeConst.Resources)).ToString(),
        Severity = DiagnosticSeverity.Warning,
        Locations =
            new[] {
                    new DiagnosticResultLocation("Test0.cs", line, column)
                }
    };

    VerifyCSharpDiagnostic(test, expected);

    VerifyCSharpFix(test, fixTest);
}
```

Afortunadamente, todos los errores anteriores se pueden tratar con las mismas técnicas que acaba de aprender.

Para corregir el primer error, primero abra **DiagnosticAnalyzer.cs** y busque el bucle foreach donde cada uno de los inicializadores de la declaración local se comprueba para asegurarse de que se les asignan valores constantes. Inmediatamente *antes* del primer bucle foreach, llame a `context.SemanticModel.GetTypeInfo()` para recuperar información detallada sobre el tipo declarado de la declaración local:

```
var variableTypeName = localDeclaration.Declaration.Type;
var variableType = context.SemanticModel.GetTypeInfo(variableTypeName).ConvertedType;
```

Después, dentro del bucle `foreach`, compruebe cada inicializador para asegurarse de que se puede convertir al tipo de variable. Agregue la siguiente comprobación después de asegurarse de que el inicializador es una constante:

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
var conversion = context.SemanticModel.ClassifyConversion(initializer.Value, variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

El siguiente cambio se basa en el último. Antes de cerrar la llave del primer bucle foreach, agregue el código siguiente para comprobar el tipo de declaración local cuando la constante es una cadena o null.

```
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
//    be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

Debe escribir algo más de código en el proveedor de corrección de código para reemplazar la palabra clave var' por el nombre de tipo correcto. Vuelva a **CodeFixProvider.cs**. El código que se va a agregar realiza los pasos siguientes:

- Compruebe si la declaración es una declaración `var` y, en su caso:
- Cree un tipo para el tipo deducido.
- Asegúrese de que la declaración de tipo no es un alias. Si es así, es válido declarar `const var`.
- Asegúrese de que `var` no es un nombre de tipo en este programa. (Si es así, `const var` es válido).
- Simplificación del nombre de tipo completo

Parece mucho código. Pero no lo es. Reemplace la línea que declara e inicializa `newLocal` con el código siguiente. Va inmediatamente después de la inicialización de `newModifiers`:

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
var variableDeclaration = localDeclaration.Declaration;
var variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    var semanticModel = await document.GetSemanticModelAsync(cancellationToken);

    // Special case: Ensure that 'var' isn't actually an alias to another type
    // (e.g. using var = System.String).
    var aliasInfo = semanticModel.GetAliasInfo(variableTypeName);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        var type = semanticModel.GetTypeInfo(variableTypeName).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named 'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            var typeName = SyntaxFactory.ParseTypeName(type.ToDisplayString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            var simplifiedTypeName = typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration = variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
var newLocal = trimmedLocal.WithModifiers(newModifiers)
                           .WithDeclaration(variableDeclaration);
```

Deberá agregar una directiva `using` para usar el tipo Simplifier:

```
using Microsoft.CodeAnalysis.Simplification;
```

Ejecute las pruebas, y todas deberían pasar. Felicítese por ejecutar el analizador terminado. Presione Ctrl + F5 para ejecutar el proyecto de analizador en una segunda instancia de Visual Studio con la extensión de la versión preliminar de Roslyn cargada.

- En la segunda instancia de Visual Studio, cree un proyecto de aplicación de consola de C# y agregue `int x = "abc";` al método Main. Gracias a la primera corrección de errores, no se debe notificar ninguna advertencia para esta declaración de variable local (aunque hay un error del compilador según lo esperado).
- A continuación, agregue `object s = "abc";` al método Main. Debido a la segunda corrección de errores, no se debe notificar ninguna advertencia.
- Por último, agregue otra variable local que usa la palabra clave `var`. Observará que se notifica una advertencia y que aparece una sugerencia debajo a la izquierda.
- Mueva el símbolo de intercalación del editor sobre el subrayado ondulado y presione Ctrl+ para mostrar la corrección de código sugerida. Al seleccionar la corrección de código, tenga en cuenta que la palabra clave var' ahora se controla correctamente.

Por último, agregue el código siguiente:

```
int i = 2;
int j = 32;
int k = i + j;
```

Después de estos cambios, obtendrá un subrayado ondulado rojo solo en las dos primeras variables. Agregue `const` a `i` y `j`, y obtendrá una nueva advertencia sobre `k` porque ahora puede ser `const`.

¡Enhorabuena! Ha creado su primera extensión de .NET Compiler Platform que realiza un análisis de código sobre la marcha para detectar un problema y proporciona una solución rápida para corregirlo. Durante el proceso, ha aprendido muchas de las API de código que forman parte del SDK de .NET Compiler Platform (API de Roslyn). Puede comprobar su trabajo con el ejemplo completo en nuestro repositorio de ejemplos de GitHub.

Otros recursos

- Introducción al análisis de sintaxis
- Introducción al análisis semántico

Guía de programación de C#

29/06/2020 • 2 minutes to read • Edit Online

Esta sección proporciona información detallada sobre las funcionalidades y características claves del lenguaje C# a las que C# puede acceder a través de .NET.

En la mayor parte de esta sección se supone que ya sabe algo sobre C# y que conoce los conceptos de programación generales. Si nunca ha programado ni ha trabajado con C#, puede visitar los tutoriales de Introducción a C# o el tutorial de .NET en explorador, que no requiere ningún conocimiento previo de programación.

Para obtener información sobre determinadas palabras clave, operadores y directivas de preprocesador, vea Referencia de C#. Para obtener información sobre la especificación del lenguaje C#, vea Especificación del lenguaje C#.

Secciones de programa

Dentro de un programa de C#

Main() y argumentos de la línea de comandos

Secciones de lenguaje

Instrucciones, expresiones y operadores

Tipos

Clases y structs

Interfaces

Delegados

Matrices

Cadenas

Propiedades

Indizadores

Eventos

Genéricos

Iteradores

Expresiones de consulta LINQ

Expresiones lambda

Espacios de nombres

Código no seguro y punteros

Comentarios de documentación XML

Secciones de la plataforma

Dominios de aplicación

Ensamblados de .NET

Atributos

Colecciones

Excepciones y control de excepciones

Registro y sistema de archivos (Guía de programación de C#)

Interoperabilidad

Reflexión

Vea también

- Referencia de C#

Dentro de un programa de C#

18/03/2020 • 2 minutes to read • Edit Online

En esta sección, se describe la estructura general de un programa de C# y se incluye el ejemplo estándar "Hola mundo".

En esta sección

- Hello World, su primer programa
- Estructura general de un programa de C#
- Nombres de identificador
- Convenciones de código de C#

Secciones relacionadas

- Introducción a C#
- Guía de programación de C#
- Referencia de C#
- Ejemplos y tutoriales

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#

Hola mundo, su primer programa

29/06/2020 • 8 minutes to read • Edit Online

En este artículo, usará Visual Studio para crear el tradicional programa "¡Hola mundo!" . Visual Studio es un entorno de desarrollo integrado (IDE) profesional con muchas características diseñadas para el desarrollo de .NET. Para crear este programa solo usará algunas de las características de Visual Studio. Para más información sobre Visual Studio, vea Introducción a Visual C#.

> **NOTE**
>
> Es posible que el equipo muestre nombres o ubicaciones diferentes para algunos de los elementos de la interfaz de usuario de Visual Studio en las siguientes instrucciones. La edición de Visual Studio que se tenga y la configuración que se utilice determinan estos elementos. Para obtener más información, vea Personalizar el IDE.

Creación de una aplicación

- Windows
- macOS

Inicie Visual Studio. Verá la siguiente imagen en Windows:

Seleccione **Crear un nuevo proyecto** en la esquina inferior derecha de la imagen. Visual Studio muestra el cuadro de diálogo **Nuevo proyecto**:

> **NOTE**
>
> Si es la primera vez que inicia Visual Studio, la lista de **plantillas de proyecto recientes** estará vacía.

En el cuadro de diálogo de nuevo proyecto, elija "Aplicación de consola (.NET Core)" y luego presione **Siguiente**. Asigne un nombre al proyecto, como "HolaMundo" y, después, presione **Crear**.

Visual Studio abre el proyecto. Ya es un ejemplo básico de "Hola mundo". Presione `Ctrl` + `F5` para ejecutar el proyecto. Visual Studio compila el proyecto y convierte el código fuente en un archivo ejecutable. Después, inicia una ventana de comandos que ejecuta la nueva aplicación. Debería mostrarse el texto siguiente en la ventana:

```
Hello World!

C:\Program Files\dotnet\dotnet.exe (process 11964) exited with code 0.
Press any key to close this window . . .
```

Presione cualquier tecla para cerrar la ventana.

Elementos de un programa en C#

Examinemos las partes importantes de este programa. La primera línea contiene un comentario. Los caracteres `//` convierten el resto de la línea en un comentario.

```
// A Hello World! program in C#.
```

También puede convertir un bloque de texto en un comentario escribiéndolo entre los caracteres `/*` y `*/`. Esta implementación se muestra en el ejemplo siguiente.

```
/* A "Hello World!" program in C#.
This program displays the string "Hello World!" on the screen. */
```

Una aplicación de la consola de C# debe contener un método `Main`, en el que se inicia y finaliza un control. El método `Main` es donde se crean los objetos y se ejecutan otros métodos.

El método `Main` es un método estático que reside dentro de una clase o de un struct. En el ejemplo anterior de "Hello World!", reside en una clase denominada `Hello`. Puede declarar el método `Main` de una de las siguientes maneras:

- Puede devolver `void`. Esto significa que el programa no devuelve un valor.

```
static void Main()
{
    //...
}
```

- También puede devolver un entero. El entero es el **código de salida** de la aplicación.

```
static int Main()
{
    //...
    return 0;
}
```

- Puede tomar argumentos con cualquiera de los tipos de valor devueltos.

```
static void Main(string[] args)
{
    //...
}
```

o bien

```
static int Main(string[] args)
{
    //...
    return 0;
}
```

El parámetro del método `Main`, `args`, es una matriz `string` que contiene los argumentos de la línea de comandos usados para invocar el programa.

Para obtener más información sobre cómo usar los argumentos de la línea de comandos, vea los ejemplos de Main() y argumentos de línea de comandos.

Entrada y salida

Los programas de C# suelen usar los servicios de entrada y salida proporcionados por la biblioteca en tiempo de ejecución de .NET. La instrucción `System.Console.WriteLine("Hello World!");` utiliza el método WriteLine. Este es uno de los métodos de salida de la clase Console en la biblioteca de tiempo de ejecución. Muestra el parámetro de cadena en la secuencia de salida estándar, seguida de una nueva línea. Hay otros métodos Console disponibles para las distintas operaciones de entrada y salida. Si incluye la directiva `using System;` al inicio del programa, puede usar directamente las clases y los métodos System sin calificarlos totalmente. Por ejemplo, puede llamar a

`Console.WriteLine` en vez de llamar a `System.Console.WriteLine`:

```
using System;
```

```
Console.WriteLine("Hello World!");
```

Para obtener más información sobre los métodos de entrada y salida, vea System.IO.

Vea también

- Guía de programación de C#
- Ejemplos y tutoriales
- Main() y argumentos de la línea de comandos
- Introducción a Visual C#

Estructura general de un programa de C# (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Los programas de C# pueden constar de uno o más archivos. Cada archivo puede contener cero o más espacios de nombres. Un espacio de nombres puede contener tipos como clases, structs, interfaces, enumeraciones y delegados, además de otros espacios de nombres. El siguiente es el esqueleto de un programa de C# que contiene todos estos elementos.

```
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

Secciones relacionadas

Para obtener más información:

- Clases
- Estructuras
- Espacios de nombres

- Interfaces
- Delegados

Especificación del lenguaje C#

Para obtener más información, consulte la sección Conceptos básicos de Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Dentro de un programa de C#
- Referencia de C#

Nombres de identificador

18/03/2020 • 2 minutes to read • Edit Online

Un **identificador** es el nombre que se asigna a un tipo (clase, interfaz, struct, delegado o enumeración), miembro, variable o espacio de nombres. Los identificadores válidos deben seguir estas reglas:

- Los identificadores deben comenzar con una letra, o `_`.
- Los identificadores pueden contener caracteres de letra Unicode, caracteres de dígito decimales, caracteres de conexión Unicode, caracteres de combinación Unicode o caracteres de formato Unicode. Para obtener más información sobre las categorías Unicode, vea la base de datos de categorías Unicode. Puede declarar identificadores que coincidan con palabras clave de C# mediante el prefijo `@` en el identificador. `@` no forma parte del nombre de identificador. Por ejemplo, `@if` declara un identificador denominado `if`. Estos identificadores textuales son principalmente para la interoperabilidad con los identificadores declarados en otros lenguajes.

Para obtener una definición completa de identificadores válidos, vea el tema Identificadores en la Especificación del lenguaje C#.

Convenciones de nomenclatura

Además de las reglas, hay una serie de convenciones de nomenclatura de los identificadores que se usa en las API. de NET. Por convención, los programas de C# usan `PascalCase` para nombres de tipo, espacios de nombres y todos los miembros públicos. Además, son comunes las convenciones siguientes:

- Los nombres de interfaz empiezan por una `I` mayúscula.
- Los tipos de atributo terminan con la palabra `Attribute`.
- Los tipos de enumeración usan un sustantivo singular para los que no son marcas y uno plural para los que sí.
- Los identificadores no deberían contener dos caracteres `_` consecutivos. Esos nombres están reservados para los identificadores generados por el compilador.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Dentro de un programa de C#
- Referencia de C#
- Clases
- Tipos de estructura
- Espacios de nombres
- Interfaces
- Delegados

Convenciones de código de C# (Guía de programación de C#)

18/03/2020 • 14 minutes to read • Edit Online

Las convenciones de codificación tienen los objetivos siguientes:

- Crean una apariencia coherente en el código, para que los lectores puedan centrarse en el contenido, no en el diseño.
- Permiten a los lectores comprender el código más rápidamente al hacer suposiciones basadas en la experiencia anterior.
- Facilitan la copia, el cambio y el mantenimiento del código.
- Muestran los procedimientos recomendados de C#.

Microsoft usa las instrucciones de este artículo para desarrollar ejemplos y documentación.

Convenciones de nomenclatura

- En ejemplos breves que no incluyen directivas using, use calificaciones de espacio de nombres. Si sabe que un espacio de nombres se importa en un proyecto de forma predeterminada, no es necesario completar los nombres de ese espacio de nombres. Los nombres completos pueden partirse después de un punto (.) si son demasiado largos para una sola línea, como se muestra en el ejemplo siguiente.

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- No es necesario cambiar los nombres de objetos que se crearon con las herramientas del diseñador de Visual Studio para que se ajusten a otras directrices.

Convenciones de diseño

Un buen diseño utiliza un formato que destaque la estructura del código y haga que el código sea más fácil de leer. Las muestras y ejemplos de Microsoft cumplen las convenciones siguientes:

- Utilice la configuración del Editor de código predeterminada (sangría automática, sangrías de 4 caracteres, tabulaciones guardadas como espacios). Para obtener más información, vea Opciones, editor de texto, C#, formato.
- Escriba solo una instrucción por línea.
- Escriba solo una declaración por línea.
- Si a las líneas de continuación no se les aplica sangría automáticamente, hágalo con una tabulación (cuatro espacios).
- Agregue al menos una línea en blanco entre las definiciones de método y las de propiedad.
- Utilice paréntesis para que las cláusulas de una expresión sean evidentes, como se muestra en el código siguiente.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Convenciones de los comentarios

- Coloque el comentario en una línea independiente, no al final de una línea de código.
- Comience el texto del comentario con una letra mayúscula.
- Finalice el texto del comentario con un punto.
- Inserte un espacio entre el delimitador de comentario (//) y el texto del comentario, como se muestra en el ejemplo siguiente.

```
// The following declaration creates a query. It does not run
// the query.
```

- No cree bloques con formato de asteriscos alrededor de comentarios.

Convenciones de lenguaje

En las secciones siguientes se describen las prácticas que sigue el equipo C# para preparar las muestras y ejemplos de código.

String (Tipo de datos)

- Use interpolación de cadenas para concatenar cadenas cortas, como se muestra en el código siguiente.

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- Para anexar cadenas en bucles, especialmente cuando se trabaja con grandes cantidades de texto, utilice un objeto StringBuilder.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

Variables locales con asignación implícita de tipos

- Use tipos implícitos para las variables locales cuando el tipo de la variable sea obvio desde el lado derecho de la asignación, o cuando el tipo exacto no sea importante.

```
// When the type of a variable is clear from the context, use var
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```

- No use var cuando el tipo no sea evidente desde el lado derecho de la asignación.

```
// When the type of a variable is not clear from the context, use an
// explicit type.
int var4 = ExampleClass.ResultSoFar();
```

- No confíe en el nombre de variable para especificar el tipo de la variable. Puede no ser correcto.

```
// Naming the following variable inputInt is misleading.
// It is a string.
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Evite el uso de `var` en lugar de dynamic.
- Use tipos implícitos para determinar el tipo de la variable de bucle en bucles for.

  En el ejemplo siguiente se usan tipos implícitos en una instrucción `for`.

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- No use tipos implícitos para determinar el tipo de la variable de bucle en bucles foreach.

  En el ejemplo siguiente se usan tipos explícitos en una instrucción `foreach`.

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

> **NOTE**
>
> Tenga cuidado de no cambiar accidentalmente un tipo de elemento de la colección iterable. Por ejemplo, es fácil cambiar de System.Linq.IQueryable a System.Collections.IEnumerable en una instrucción `foreach`, lo cual cambia la ejecución de una consulta.

Tipo de datos sin signo

En general, utilice `int` en lugar de tipos sin signo. El uso de `int` es común en todo C#, y es más fácil interactuar con otras bibliotecas cuando se usa `int`.

Matrices

Utilice sintaxis concisa para inicializar las matrices en la línea de declaración.

```
// Preferred syntax. Note that you cannot use var here instead of string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a time.
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

Delegados

Utilice sintaxis concisa para crear instancias de un tipo de delegado.

```
// First, in class Program, define the delegate type and a method that
// has a matching signature.

// Define the type.
public delegate void Del(string message);

// Define a method that has a matching signature.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

```
// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.
Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.
Del exampleDel1 = new Del(DelMethod);
```

Instrucciones try-catch y using en el control de excepciones

- Use una instrucción try-catch en la mayoría de casos de control de excepciones.

  ```
  static string GetValueFromArray(string[] array, int index)
  {
      try
      {
          return array[index];
      }
      catch (System.IndexOutOfRangeException ex)
      {
          Console.WriteLine("Index is out of range: {0}", index);
          throw;
      }
  }
  ```

- Simplifique el código mediante la instrucción using de C#. Si tiene una instrucción try-finally en la que el único código del bloque `finally` es una llamada al método Dispose, use en su lugar una instrucción `using`.

```
// This try-finally statement only calls Dispose in the finally block.
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}

// You can do the same thing with a using statement.
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

Operadores && y ||

Para evitar excepciones y aumentar el rendimiento omitiendo las comparaciones innecesarias, use `&&` en lugar de `&` y `||` en lugar de `|` cuando realice comparaciones, como se muestra en el ejemplo siguiente.

```
Console.Write("Enter a dividend: ");
var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following condition
// causes a run-time error. The && operator short circuits when the
// first expression is false. That is, it does not evaluate the
// second expression. The & operator evaluates both, and causes
// a run-time error when divisor is 0.
if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

New (Operador)

- Utilice la forma concisa de la creación de instancias de objeto con tipos implícitos, como se muestra en la siguiente declaración.

  ```
  var instance1 = new ExampleClass();
  ```

  La línea anterior es equivalente a la siguiente declaración.

  ```
  ExampleClass instance2 = new ExampleClass();
  ```

- Utilice inicializadores de objeto para simplificar la creación de objetos.

```
// Object initializer.
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

Control de eventos

Si va a definir un controlador de eventos que no es necesario quitar más tarde, utilice una expresión lambda.

```
public Form2()
{
    // You can use a lambda expression to define an event handler.
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

```
// Using a lambda expression shortens the following traditional definition.
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

Miembros estáticos

Llame a miembros estáticos con el nombre de clase *ClassName.StaticMember*. Esta práctica hace que el código sea más legible al clarificar el acceso estático. No califique un miembro estático definido en una clase base con el nombre de una clase derivada. Mientras el código se compila, su legibilidad se presta a confusión, y puede interrumpirse en el futuro si se agrega a un miembro estático con el mismo nombre a la clase derivada.

Consultas LINQ

- Utilice nombres descriptivos para las variables de consulta. En el ejemplo siguiente, se utiliza `seattleCustomers` para los clientes que se encuentran en Seattle.

  ```
  var seattleCustomers = from customer in customers
                         where customer.City == "Seattle"
                         select customer.Name;
  ```

- Utilice alias para asegurarse de que los nombres de propiedad de tipos anónimos se escriben correctamente con mayúscula o minúscula, usando para ello la grafía Pascal.

  ```
  var localDistributors =
      from customer in customers
      join distributor in distributors on customer.City equals distributor.City
      select new { Customer = customer, Distributor = distributor };
  ```

- Cambie el nombre de las propiedades cuando puedan ser ambiguos en el resultado. Por ejemplo, si la consulta devuelve un nombre de cliente y un identificador de distribuidor, en lugar de dejarlos como `Name` e `ID` en el resultado, cambie su nombre para aclarar que `Name` es el nombre de un cliente e `ID` es el identificador de un distribuidor.

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals distributor.City
    select new { CustomerName = customer.Name, DistributorID = distributor.ID };
```

- Utilice tipos implícitos en la declaración de variables de consulta y variables de intervalo.

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Alinee las cláusulas de consulta bajo la cláusula from, como se muestra en los ejemplos anteriores.
- Use cláusulas where antes de otras cláusulas de consulta para asegurarse de que las cláusulas de consulta posteriores operan en un conjunto de datos reducido y filtrado.

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Use varias cláusulas `from` en lugar de una cláusula join para obtener acceso a colecciones internas. Por ejemplo, una colección de objetos `Student` podría contener cada uno un conjunto de resultados de exámenes. Cuando se ejecuta la siguiente consulta, devuelve cada resultado superior a 90, además del apellido del alumno que recibió la puntuación.

```
// Use a compound from to access the inner sequence within each element.
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

Seguridad

Siga las instrucciones de Instrucciones de codificación segura.

Vea también

- Convenciones de código de Visual Basic
- Instrucciones de codificación segura

Main() y argumentos de la línea de comandos (Guía de programación de C#)

29/06/2020 • 4 minutes to read • Edit Online

El método `Main` es el punto de entrada de una aplicación de C# (las bibliotecas y los servicios no requieren un método `Main` como punto de entrada). Cuando se inicia la aplicación, el método `Main` es el primero que se invoca.

Solo puede haber un punto de entrada en un programa de C#. Si hay más de una clase que tiene un método `Main`, debe compilar el programa con la opción del compilador `-main` para especificar qué método `Main` desea utilizar como punto de entrada. Para obtener más información, vea -main (Opciones del compilador de C#).

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

Información general

- El método `Main` es el punto de entrada de un programa ejecutable; es donde se inicia y finaliza el control del programa.
- `Main` se declara dentro de una clase o estructura. El valor de `Main` debe ser estático y no público. (En el ejemplo anterior, recibe el acceso predeterminado de privado). La clase o estructura envolvente no debe ser estático.
- `Main` puede tener un tipo de valor devuelto `void`, `int` o, a partir de C# 7.1, `Task` o `Task<int>`.
- Solo si `Main` devuelve un tipo de valor devuelto `Task` o `Task<int>`, la declaración de `Main` puede incluir el modificador `async`, pero tenga en cuenta que se excluirá de forma específica un método `async void Main`.
- El método `Main` se puede declarar con o sin un parámetro `string[]` que contiene los argumentos de línea de comandos. Al usar Visual Studio para crear aplicaciones de Windows, se puede agregar el parámetro manualmente o usar el método GetCommandLineArgs() con el fin de obtener los argumentos de la línea de comandos. Los parámetros se leen como argumentos de línea de comandos indizados con cero. A diferencia de C y C++, el nombre del programa no se trata como el primer argumento de línea de comandos en la matriz `args`, pero es el primer elemento del método GetCommandLineArgs().

A continuación se muestra una lista de firmas `Main` válidas:

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

En los ejemplos anteriores se usa el modificador de acceso público. Es el procedimiento habitual, pero no es necesario.

Al agregar los tipos de valor devuelto `async`, `Task` y `Task<int>`, se simplifica el código de programa cuando las aplicaciones de consola tienen que realizar tareas de inicio y `await` de operaciones asincrónicas en `Main`.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Compilar la línea de comandos con csc.exe
- Guía de programación de C#
- Métodos
- Dentro de un programa de C#

Argumentos de la línea de comandos (Guía de programación de C#)

29/06/2020 • 5 minutes to read • Edit Online

Puede enviar argumentos al método `Main` definiéndolo de una de las siguientes maneras:

```
static int Main(string[] args)
```

```
static void Main(string[] args)
```

> **NOTE**
> Para habilitar los argumentos de la línea de comandos en el método `Main` de una aplicación Windows Forms, tendrá que modificar manualmente la firma de `Main` en *program.cs*. El código generado por el Diseñador de Windows Forms crea un `Main` sin ningún parámetro de entrada. También puede usar Environment.CommandLine o Environment.GetCommandLineArgs para tener acceso a los argumentos de la línea de comandos desde cualquier punto en una consola o aplicación Windows.

El parámetro del método `Main` es una matriz String que representa los argumentos de la línea de comandos. Normalmente, para determinar si hay argumentos, se prueba la propiedad `Length`; por ejemplo:

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

> **TIP**
> La matriz `args` no puede ser NULL, así que es seguro acceder a la propiedad `Length` sin comprobar los valores NULL.

También puede convertir los argumentos de cadena en tipos numéricos mediante la clase Convert o el método `Parse`. Por ejemplo, la siguiente instrucción convierte la `string` en un número `long` mediante el método Parse:

```
long num = Int64.Parse(args[0]);
```

También se puede usar el tipo de C# `long`, que tiene como alias `Int64`:

```
long num = long.Parse(args[0]);
```

También puede usar el método `ToInt64` de la clase `Convert` para hacer lo mismo:

```
long num = Convert.ToInt64(s);
```

Para obtener más información, vea Parse y Convert.

Ejemplo

En el ejemplo siguiente se muestra cómo usar argumentos de la línea de comandos en una aplicación de consola. La aplicación toma un argumento en tiempo de ejecución, lo convierte en un entero y calcula el factorial del número. Si no se proporciona ningún argumento, la aplicación emite un mensaje en el que se explica el uso correcto del programa.

Para compilar y ejecutar la aplicación desde un símbolo del sistema, siga estos pasos:

1. Pegue el código siguiente en cualquier editor de texto y, después, guarde el archivo como archivo de texto con el nombre *Factorial.cs*.

```
// Add a using directive for System if the directive isn't already present.

public class Functions
{
    public static long Factorial(int n)
    {
        // Test for invalid input.
        if ((n < 0) || (n > 20))
        {
            return -1;
        }

        // Calculate the factorial iteratively rather than recursively.
        long tempResult = 1;
        for (int i = 1; i <= n; i++)
        {
            tempResult *= i;
        }
        return tempResult;
    }
}

class MainClass
{
    static int Main(string[] args)
    {
        // Test if input arguments were supplied.
        if (args.Length == 0)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Try to convert the input arguments to numbers. This will throw
        // an exception if the argument is not a number.
        // num = int.Parse(args[0]);
        int num;
        bool test = int.TryParse(args[0], out num);
        if (!test)
        {
            Console.WriteLine("Please enter a numeric argument.");
            Console.WriteLine("Usage: Factorial <num>");
            return 1;
        }

        // Calculate factorial.
        long result = Functions.Factorial(num);

        // Print result.
        if (result == -1)
            Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            Console.WriteLine($"The Factorial of {num} is {result}.");

        return 0;
    }
}
// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.
```

2. En la pantalla **Inicio** o en el menú **Inicio**, abra una ventana del **símbolo del sistema para desarrolladores** de Visual Studio y navegue hasta la carpeta que contiene el archivo que acaba de crear.

3. Escriba el siguiente comando para compilar la aplicación.

```
csc Factorial.cs
```

Si la aplicación no tiene ningún error de compilación, se creará un archivo ejecutable con el nombre *Factorial.exe*.

4. Escriba el siguiente comando para calcular el factorial de 3:

```
Factorial 3
```

5. El comando genera este resultado: `The factorial of 3 is 6.`

> **NOTE**
> Al ejecutar una aplicación en Visual Studio, puede especificar argumentos de la línea de comandos en la Página Depuración, Diseñador de proyectos.

Vea también

- System.Environment
- Guía de programación de C#
- Main() y argumentos de la línea de comandos
- Procedimiento para mostrar argumentos de la línea de comandos
- Valores devueltos de Main()
- Clases

Procedimiento Mostrar argumentos de la línea de comandos (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

A los argumentos proporcionados a un archivo ejecutable en la línea de comandos se puede tener acceso a través de un parámetro opcional de `Main`. Los argumentos se proporcionan en forma de una matriz de cadenas. Cada elemento de la matriz contiene un argumento. Se quita el espacio en blanco entre los argumentos. Por ejemplo, considere estas invocaciones de línea de comandos de un ejecutable ficticio:

ENTRADA EN LA LÍNEA DE COMANDOS	MATRIZ DE CADENAS QUE SE PASA A MAIN
executable.exe a b c	"a" "b" "c"
executable.exe one two	"one" "two"
executable.exe "one two" three	"one two" "three"

> **NOTE**
> Si se ejecuta una aplicación en Visual Studio, se pueden especificar argumentos de línea de comandos en la Página Depuración, Diseñador de proyectos.

Ejemplo

En este ejemplo se muestran los argumentos de línea de comandos pasados a una aplicación de línea de comandos. La salida que se muestra corresponde a la primera entrada de la tabla anterior.

```
class CommandLine
{
    static void Main(string[] args)
    {
        // The Length property provides the number of array elements.
        Console.WriteLine($"parameter count = {args.Length}");

        for (int i = 0; i < args.Length; i++)
        {
            Console.WriteLine($"Arg[{i}] = [{args[i]}]");
        }
    }
}
/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

Vea también

- Guía de programación de C#
- Compilar la línea de comandos con csc.exe
- Main() y argumentos de la línea de comandos
- Valores devueltos de Main()

Valores devueltos de Main() (Guía de programación de C#)

16/04/2020 • 5 minutes to read • Edit Online

El método `Main` puede devolver el valor `void`:

```
static void Main()
{
    //...
}
```

También puede devolver un valor `int`:

```
static int Main()
{
    //...
    return 0;
}
```

Si el valor devuelto de `Main` no se usa, la devolución de `void` permite contar con un código ligeramente más simple. En cambio, devolver un valor entero permite que el programa comunique información de estado a otros programas o scripts que invocan el archivo ejecutable. El valor devuelto de `Main` se trata como el código de salida para el proceso. Si se devuelve `void` de `Main`, el código de salida será implícitamente `0`. En el ejemplo siguiente se muestra cómo se puede acceder al valor devuelto de `Main`.

Ejemplo

En este ejemplo se usan las herramientas de línea de comandos de .NET Core. Si no está familiarizado con las herramientas de línea de comandos de .NET Core, puede obtener información sobre ellas en este artículo de introducción.

Modifique el método `Main` en *program.cs* como se indica a continuación:

```
// Save this program as MainReturnValTest.cs.
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

Cuando un programa se ejecuta en Windows, cualquier valor devuelto por la función `Main` se almacena en una variable de entorno. Esta variable de entorno se puede recuperar mediante `ERRORLEVEL` desde un archivo por lotes, o mediante `$LastExitCode` desde PowerShell.

Puede compilar la aplicación mediante el comando `dotnet build` de la CLI de dotnet.

Después, cree un script de PowerShell para ejecutar la aplicación y mostrar el resultado. Pegue el código siguiente en un archivo de texto y guárdelo como `test.ps1` en la carpeta que contiene el proyecto. Ejecute el script de

PowerShell. Para ello, escriba `test.ps1` en el símbolo del sistema de PowerShell.

Dado que el código devuelve el valor cero, el archivo por lotes comunicará un resultado satisfactorio. En cambio, si cambia MainReturnValTest.cs para que devuelva un valor distinto de cero y luego vuelve a compilar el programa, la ejecución posterior del script de PowerShell informará de que se ha producido un error.

```
dotnet run
```

```
if ($LastExitCode -eq 0) {
    Write-Host "Execution succeeded"
} else
{
    Write-Host "Execution Failed"
}
Write-Host "Return value = " $LastExitCode
```

Resultados de ejemplo

```
Execution succeeded
Return value = 0
```

Valores devueltos asincrónicos de Main

Los valores devueltos asincrónicos de Main mueven el código reutilizable necesario para llamar a métodos asincrónicos en `Main` al código generado por el compilador. Antes, había que escribir esta construcción para llamar a código asincrónico y asegurarse de que el programa se ejecutase hasta la finalización de la operación asincrónica:

```
public static void Main()
{
    AsyncConsoleWork().GetAwaiter().GetResult();
}

private static async Task<int> AsyncConsoleWork()
{
    // Main body here
    return 0;
}
```

Ahora, esto se puede reemplazar por lo siguiente:

```
static async Task<int> Main(string[] args)
{
    return await AsyncConsoleWork();
}
```

La ventaja de la nueva sintaxis es que el compilador siempre genera el código correcto.

Código generado por el compilador

Cuando el punto de entrada de la aplicación devuelve `Task` o `Task<int>`, el compilador genera un nuevo punto de entrada que llama al método de punto de entrada declarado en el código de la aplicación. Suponiendo que este punto de entrada se denomina `$GeneratedMain`, el compilador genera el código siguiente para estos puntos de entrada:

- `static Task Main()` hace que el compilador emita el equivalente de `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task Main(string[])` hace que el compilador emita el equivalente de `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`
- `static Task<int> Main()` hace que el compilador emita el equivalente de `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`
- `static Task<int> Main(string[])` hace que el compilador emita el equivalente de `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`

> **NOTE**
>
> Si en los ejemplos se usase el modificador `async` en el método `Main`, el compilador generaría el mismo código.

Vea también

- Guía de programación de C#
- Referencia de C#
- Main() y argumentos de la línea de comandos
- Procedimiento para mostrar argumentos de la línea de comandos

Conceptos de programación (C#)

29/06/2020 • 2 minutes to read • Edit Online

En esta sección se explican los conceptos de programación del lenguaje C#.

En esta sección

TITLE	DESCRIPCIÓN
Ensamblados de .NET	Describe cómo crear y utilizar ensamblados.
Programación asincrónica con Async y Await (C#)	Describe cómo escribir soluciones asincrónicas mediante las palabras clave Async y Await en C#. Incluye un tutorial.
Atributos (C#)	Describe cómo proporcionar información adicional sobre elementos de programación como tipos, campos, métodos y propiedades mediante el uso de atributos.
Colecciones (C#)	Describe algunos de los tipos de colecciones proporcionadas por .NET. Muestra cómo usar colecciones sencillas y colecciones de pares clave-valor.
Covarianza y contravarianza (C#)	Describe cómo habilitar la conversión implícita de parámetros de tipo genérico en interfaces y delegados.
Árboles de expresión (C#)	Explica cómo puede utilizar árboles de expresión para habilitar la modificación dinámica de código ejecutable.
Iteradores (C#)	Describe los iteradores, que se usan para recorrer colecciones y devolver los elementos uno a uno.
Language Integrated Query (LINQ) (C#)	Se describen las eficaces funciones de consulta de la sintaxis del lenguaje C#, así como el modelo para consultar bases de datos relacionales, documentos XML, conjuntos de datos y colecciones en memoria.
Programación orientada a objetos (C#)	Describe conceptos comunes orientados a objetos, incluidos la encapsulación, la herencia y el polimorfismo.
Reflexión (C#)	Se explica cómo usar la reflexión para crear dinámicamente una instancia de un tipo, enlazar el tipo a un objeto existente u obtener el tipo desde un objeto existente e invocar sus métodos, o acceder a sus campos y propiedades.
Serialización (C#)	Describe los conceptos clave de la serialización binaria, XML y SOAP.

Secciones relacionadas

Sugerencias para mejorar el rendimiento	Se describen varias reglas básicas que pueden ayudarle a aumentar el rendimiento de la aplicación.

Programación asincrónica con async y await

29/06/2020 • 23 minutes to read • Edit Online

El modelo de programación asincrónica de tareas (TAP) es una abstracción del código asincrónico. El código se escribe como una secuencia de instrucciones, como es habitual. Puede leerlo como si cada instrucción se completase antes de comenzar la siguiente. El compilador realiza una serie de transformaciones, debido a que algunas de estas instrucciones podrían empezar a funcionar y devolver una clase Task que representa el trabajo en curso.

Este es el objetivo de la sintaxis: habilitar código que se lea como una secuencia de instrucciones, pero que se ejecute siguiendo un orden mucho más complicado, en función de la asignación de recursos externos y del momento en el que se completen las tareas. Es similar a la manera en la que las personas dan instrucciones para los procesos que incluyen las tareas asincrónicas. En este artículo, usará un ejemplo con instrucciones para preparar el desayuno que le ayudará a comprender cómo las palabras clave `async` y `await` facilitan el proceso de razonar sobre el código, que incluye una serie de instrucciones asincrónicas. Para explicar cómo se prepara un desayuno, probablemente escribirá unas instrucciones parecidas a las que se recogen en la lista siguiente:

1. Sirva una taza de café.
2. Caliente una sartén y fría dos huevos.
3. Fría tres lonchas de beicon.
4. Tueste dos rebanadas de pan.
5. Unte el pan con mantequilla y mermelada.
6. Sirva un vaso de zumo de naranja.

Si tiene experiencia en la cocina, lo más probable es que ejecute estas instrucciones de forma **asincrónica**. Primero, calentará la sartén para los huevos e irá friendo el beicon. Después, pondrá el pan en la tostadora y empezará a freír los huevos. En cada paso del proceso, iniciará una tarea y volverá la atención a las tareas que tiene pendientes.

La preparación del desayuno es un buen ejemplo de un trabajo asincrónico que no es paralelo. Una persona (o un subproceso) puede controlar todas estas tareas. Siguiendo con la analogía del desayuno, una persona puede preparar el desayuno asincrónicamente si comienza la tarea siguiente antes de que finalice la anterior. Los alimentos se cocinan tanto si una persona supervisa el proceso como si no. En cuanto se empieza a calentar la sartén para los huevos, se puede comenzar a freír el beicon. Una vez que el beicon se esté haciendo, se puede poner el pan en la tostadora.

En el caso de un algoritmo paralelo, necesitaría varios cocineros (o subprocesos). Uno se encargaría de los huevos, otro del beicon, etc. Cada uno de ellos se centraría en una sola tarea. Un cocinero (o subproceso) se bloqueará al esperar asincrónicamente a que el beicon se dore para darle la vuelta, o al esperar a que las tostadas estén listas.

Piense ahora en estas mismas instrucciones escritas como instrucciones de C#:

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
            Console.WriteLine("bacon is ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the toaster");
            }
            Console.WriteLine("Start toasting...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static Bacon FryBacon(int slices)
        {
            Console.WriteLine($"putting {slices} slices of bacon in the pan");
            Console.WriteLine("cooking first side of bacon...");
            Task.Delay(3000).Wait();
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("flipping a slice of bacon");
            }
            Console.WriteLine("cooking the second side of bacon...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put bacon on plate");

            return new Bacon();
        }

        private static Egg FryEggs(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            Task.Delay(3000).Wait();
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put eggs on plate");
```

```
            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

El desayuno preparado de forma sincrónica tardó unos 30 minutos porque el total es la suma de cada tarea individual.

> **NOTE**
> Las clases `Coffee`, `Egg`, `Bacon`, `Toast` y `Juice` están vacías. Simplemente son clases de marcador creadas para la demostración; no contienen propiedades y no sirven para ningún otro propósito.

Los equipos no interpretan estas instrucciones de la misma manera que las personas. El equipo se bloqueará en cada instrucción hasta que el trabajo se complete antes de pasar a la instrucción siguiente. Podría decirse que esto da lugar a un desayuno poco satisfactorio. Las tareas posteriores no se pueden iniciar mientras no se completen las anteriores. Así pues, se tardará mucho más en preparar el desayuno y algunos alimentos se habrán enfriado incluso antes de servirse.

Si quiere que el equipo ejecute las instrucciones anteriores de forma asincrónica, debe escribir código asincrónico.

Estas cuestiones son importantes para los programas que se escriben hoy en día. Al escribir programas cliente, le interesa que la interfaz de usuario responda a la entrada del usuario. La aplicación no debería hacer que un teléfono parezca congelado mientras descarga datos de la Web. Al escribir programas de servidor, no le conviene que los subprocesos se bloqueen. La intención es que puedan atender también otras solicitudes. El uso de código sincrónico cuando existen alternativas asincrónicas va en detrimento de la capacidad de escalar horizontalmente a un menor coste. Al final, los subprocesos bloqueados pasarán factura.

Las aplicaciones modernas de más éxito requieren código asincrónico. Sin la compatibilidad con los lenguajes, la escritura de código asincrónico requería devoluciones de llamada, eventos de finalización u otros medios que impedían ver claramente la intención original del código. La ventaja del código sincrónico es que resulta fácil de entender. Las acciones detalladas hacen que sea fácil examinar el código y comprenderlo. Los modelos asincrónicos tradicionales obligaban a centrarse en la naturaleza asincrónica del código, no en las acciones fundamentales.

Uso de await para evitar los bloqueos

El código anterior muestra una práctica incorrecta, que consiste en construir código sincrónico para llevar a cabo operaciones asincrónicas. Tal como está escrito, este código bloquea el subproceso que lo ejecuta y le impide realizar cualquier otro trabajo. Nada lo interrumpirá mientras alguna de las tareas esté en curso. Es como si usted se quedara mirando la tostadora después de meter el pan y no pudiera oír a nadie que le dirigiera la palabra hasta que las tostadas estuvieran listas.

Empecemos por actualizar este código para que el subproceso no se bloquee mientras se ejecutan las tareas. La palabra clave `await` proporciona un modo sin bloqueo para iniciar una tarea y, después, proseguir la ejecución cuando dicha tarea se complete. Una versión asincrónica sencilla del código para preparar el desayuno tendría un aspecto parecido al del fragmento siguiente:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

IMPORTANT

El tiempo total transcurrido es aproximadamente el mismo que el de la versión inicial sincrónica. El código todavía tiene que aprovechar algunas de las características clave de programación asincrónica.

TIP

Los cuerpos del método de `FryEggsAsync`, `FryBaconAsync` y `ToastBreadAsync` se han actualizado para devolver `Task<Egg>`, `Task<Bacon>` y `Task<Toast>`, respectivamente. Se cambia el nombre de los métodos de su versión original para incluir el sufijo "Async". Sus implementaciones se muestran como parte de la versión final más adelante en este artículo.

Este código no produce un bloqueo mientras se cocinan los huevos o el beicon, pero tampoco inicia otras tareas. Es decir, pondría el pan en la tostadora y se quedaría esperando a que estuviera listo, pero, por lo menos, si alguien reclamara su atención, le haría caso. En un restaurante en el que se atienden varios pedidos,

el cocinero empezaría a preparar otro desayuno mientras se hace el primero.

El subproceso que se encarga del desayuno ya no se bloquearía mientras espera por las tareas iniciadas que aún no han terminado. En algunas aplicaciones, lo único que se necesita es este cambio. Una aplicación de interfaz gráfica de usuario seguirá respondiendo al usuario con este único cambio. Aun así, para este escenario, necesita algo más. No le interesa que todas las tareas de componente se ejecuten secuencialmente. Es mejor iniciar cada una de estas tareas sin esperar a que la tarea anterior se complete.

Inicio simultáneo de tareas

En muchos escenarios, necesita iniciar varias tareas independientes de inmediato. Después, a medida que finalice cada tarea, podrá seguir con el trabajo que esté listo. Siguiendo con la analogía del desayuno, esta es la manera más rápida de prepararlo. Además, de este modo, todo estará listo aproximadamente en el mismo momento. Así podrá disfrutar de un desayuno caliente.

La clase System.Threading.Tasks.Task y los tipos relacionados se pueden usar para razonar sobre las tareas que están en curso. Esto le permite escribir código que se asemeje más a la manera en que realmente se prepara el desayuno. Para ello, cocinará los huevos, el beicon y las tostadas al mismo tiempo. Como cada uno de estos elementos requiere una acción, se ocupará de esa tarea, se encargará de la acción siguiente y esperará por todo lo que necesite su atención.

Comenzará una tarea y conservará el objeto Task que representa el trabajo. Llevará a cabo una instrucción `await` para esperar por cada tarea antes de trabajar con su resultado.

Realicemos estos cambios en el código del desayuno. El primer paso consiste en almacenar las tareas de las operaciones cuando se inician, en lugar de esperar por ellas:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");

Juice oj = PourOJ();
Console.WriteLine("oj is ready");
Console.WriteLine("Breakfast is ready!");
```

Después, puede mover las instrucciones `await` del beicon y los huevos al final del método, antes de servir el desayuno:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Console.WriteLine("Breakfast is ready!");
```

El desayuno preparado de forma asincrónica tardó unos 20 minutos porque algunas tareas pudieron efectuarse simultáneamente.

El código anterior funciona mejor. Iniciará todas las tareas asincrónicas a la vez y esperará por una tarea solo cuando necesite los resultados. El código anterior se parece al código de una aplicación web que realiza solicitudes a diferentes microservicios y, después, combina los resultados en una sola página. Podrá realizar todas las solicitudes de inmediato y, luego, llevará a cabo una instrucción `await` para esperar por todas esas tareas y componer la página web.

Composición con tareas

Ya tiene todo listo al mismo tiempo para el desayuno, excepto las tostadas. La preparación de las tostadas es la composición de una operación asincrónica (tostar el pan) y varias operaciones sincrónicas (untar la mantequilla y la mermelada). La actualización de este código ilustra un concepto importante:

> **IMPORTANT**
> La composición de una operación asincrónica seguida de un trabajo sincrónico es una operación asincrónica. Dicho de otra manera, si una parte cualquiera de una operación es asincrónica, toda la operación es asincrónica.

En el código anterior se muestra que se puede usar un objeto Task o Task<TResult> para conservar tareas en ejecución. Lleva a cabo una instrucción `await` para esperar por una tarea a fin de poder usar su resultado. El siguiente paso consiste en crear métodos que representan la combinación de otro trabajo. Antes de servir el desayuno, quiere esperar por la tarea que representa tostar el pan antes de untar la mantequilla y la mermelada. Puede representar este trabajo con el código siguiente:

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

El método anterior tiene el modificador `async` en su firma, lo que indica al compilador que este método incluye una instrucción `await`, es decir, que contiene operaciones asincrónicas. Este método representa la tarea que tuesta el pan y, después, agrega la mantequilla y la mermelada. El método devuelve un objeto Task<TResult> que representa la composición de estas tres operaciones. El bloque principal de código se convierte en lo siguiente:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

El cambio anterior ilustra una técnica importante para trabajar con código asincrónico. Para componer tareas, las operaciones se separan en un método nuevo que devuelve una tarea. Usted puede elegir cuándo se debe esperar por esta tarea y puede iniciar otras tareas simultáneamente.

Espera de la finalización de las tareas de forma eficaz

La serie de instrucciones `await` al final del código anterior se puede mejorar mediante el uso de métodos de la clase `Task`. Una de estas API es WhenAll, que devuelve un objeto Task que se completa cuando finalizan

todas las tareas de la lista de argumentos, como se muestra en el código siguiente:

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("eggs are ready");
Console.WriteLine("bacon is ready");
Console.WriteLine("toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Otra opción consiste en usar WhenAny, que devuelve un objeto `Task<Task>` que se completa cuando finaliza cualquiera de sus argumentos. Puede esperar por la tarea devuelta, con la certeza de saber que ya ha terminado. En el código siguiente se muestra cómo se puede usar WhenAny para esperar a que la primera tarea finalice y, después, procesar su resultado. Después de procesar el resultado de la tarea completada, quítela de la lista de tareas que se pasan a `WhenAny`.

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("bacon is ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("toast is ready");
    }
    breakfastTasks.Remove(finishedTask);
}
```

Después de todos estos cambios, la versión final del código tiene un aspecto similar al siguiente: .

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    class Program
    {
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var baconTask = FryBaconAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == baconTask)
                {
                    Console.WriteLine("bacon is ready");
```

```
            }
            else if (finishedTask == toastTask)
            {
                Console.WriteLine("toast is ready");
            }
            breakfastTasks.Remove(finishedTask);
        }

        Juice oj = PourOJ();
        Console.WriteLine("oj is ready");
        Console.WriteLine("Breakfast is ready!");
    }

    static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
    {
        var toast = await ToastBreadAsync(number);
        ApplyButter(toast);
        ApplyJam(toast);

        return toast;
    }

    private static Juice PourOJ()
    {
        Console.WriteLine("Pouring orange juice");
        return new Juice();
    }

    private static void ApplyJam(Toast toast) =>
        Console.WriteLine("Putting jam on the toast");

    private static void ApplyButter(Toast toast) =>
        Console.WriteLine("Putting butter on the toast");

    private static async Task<Toast> ToastBreadAsync(int slices)
    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the toaster");
        }
        Console.WriteLine("Start toasting...");
        await Task.Delay(3000);
        Console.WriteLine("Remove toast from toaster");

        return new Toast();
    }

    private static async Task<Bacon> FryBaconAsync(int slices)
    {
        Console.WriteLine($"putting {slices} slices of bacon in the pan");
        Console.WriteLine("cooking first side of bacon...");
        await Task.Delay(3000);
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("flipping a slice of bacon");
        }
        Console.WriteLine("cooking the second side of bacon...");
        await Task.Delay(3000);
        Console.WriteLine("Put bacon on plate");

        return new Bacon();
    }

    private static async Task<Egg> FryEggsAsync(int howMany)
    {
        Console.WriteLine("Warming the egg pan...");
        await Task.Delay(3000);
        Console.WriteLine($"cracking {howMany} eggs");
        Console.WriteLine("cooking the eggs ...");
```

```
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

La versión final del desayuno preparado de forma asincrónica tardó aproximadamente 15 minutes porque algunas tareas pudieron efectuarse simultáneamente y el código pudo supervisar varias tareas a la vez, tomando medidas solo en caso necesario.

Este código final es asincrónico. Refleja con más precisión la manera en que una persona prepara un desayuno. Compare el código anterior con el primer ejemplo de código del artículo. Las acciones principales siguen siendo claras cuando se lee el código. De hecho, puede leerlo como si se tratara de las instrucciones para preparar el desayuno que se indican al principio del artículo. Las características del lenguaje para `async` y `await` proporcionan la traducción que cualquier persona haría para seguir las instrucciones escritas, a saber: las tareas deben iniciarse a medida que sea posible y debe evitarse el bloqueo por esperar a que se completen las tareas.

Pasos siguientes

Información sobre el modelo de programación asincrónica de tareas

Modelo de programación asincrónica de tareas

29/06/2020 • 33 minutes to read • Edit Online

Puede evitar cuellos de botella de rendimiento y mejorar la capacidad de respuesta total de la aplicación mediante la programación asincrónica. Sin embargo, las técnicas tradicionales para escribir aplicaciones asincrónicas pueden resultar complicadas, haciéndolas difícil de escribir, depurar y mantener.

C# 5 ha introducido un enfoque simplificado, la programación asincrónica, que aprovecha la compatibilidad asincrónica de .NET Framework 4.5 y versiones posteriores, NET Core y Windows Runtime. El compilador realiza el trabajo difícil que el desarrollador suele realizar y la aplicación conserva una estructura lógica similar al código sincrónico. Como resultado, se obtienen todas las ventajas de la programación asincrónica con una parte del trabajo.

Este tema proporciona información general sobre cuándo y cómo utilizar la programación asincrónica e incluye vínculos que admiten temas con detalles y ejemplos.

La asincronía mejora la capacidad de respuesta

La asincronía es esencial para actividades que pueden producir bloqueos, por ejemplo, el acceso a Internet. Tener acceso a un recurso web a veces es lento o va retrasado. Si tal actividad queda bloqueada en un proceso sincrónico, toda la aplicación deberá esperar. En un proceso asincrónico, la aplicación puede continuar con otro trabajo que no dependa del recurso web hasta que finaliza la tarea que puede producir bloqueos.

En la tabla siguiente se muestran las áreas típicas donde la programación asincrónica mejora su capacidad de respuesta. Las API enumeradas desde .NET y Windows Runtime contienen métodos que admiten la programación asincrónica.

ÁREA DE APLICACIÓN	TIPOS DE .NET CON MÉTODOS ASINCRÓNICOS	TIPOS DE WINDOWS RUNTIME CON MÉTODOS ASINCRÓNICOS
Acceso web	HttpClient	SyndicationClient
Trabajar con archivos	StreamWriter, StreamReader, XmlReader	StorageFile
Trabajar con imágenes		MediaCapture, BitmapEncoder, BitmapDecoder
Programar WCF	Operaciones sincrónicas y asincrónicas	

La asincronía es especialmente valiosa para aquellas aplicaciones que obtienen acceso al subproceso de interfaz de usuario, ya que todas las actividades relacionadas con la interfaz de usuario normalmente comparten un único subproceso. Si se bloquea un proceso en una aplicación sincrónica, se bloquean todos. La aplicación deja de responder y puede que se piense que se ha producido un error cuando en realidad la aplicación está esperando.

Cuando se usan métodos asincrónicos, la aplicación continúa respondiendo a la interfaz de usuario. Puede cambiar el tamaño o minimizar una ventana, por ejemplo, o puede cerrar la aplicación si no desea esperar a que finalice.

El enfoque basado en asincrónico agrega el equivalente de una transmisión automática a la lista de opciones entre las que puede elegir al diseñar operaciones asincrónicas. Es decir, obtiene todas las ventajas de la programación asincrónica tradicional pero con mucho menos trabajo de desarrollador.

Los métodos asincrónicos son más fáciles de escribir

Las palabras clave async y await en C# son fundamentales en la programación asincrónica. Con esas dos palabras clave, se pueden usar los recursos de .NET Framework, .NET Core o Windows Runtime para crear un método asincrónico casi tan fácilmente como se crea un método sincrónico. Los métodos asincrónicos que se definen mediante la palabra clave `async` se denominan *métodos asincrónicos*.

En el ejemplo siguiente se muestra un método asincrónico. Prácticamente todos los elementos del código deberían resultarle familiares.

Puede encontrar un archivo de ejemplo completo de Windows Presentation Foundation (WPF) al final de este tema y puede descargar el ejemplo de Async Sample: Example from "Asynchronous Programming with Async and Await" (Ejemplo de async: Ejemplo de "Programación asincrónica con Async y Await).

```
async Task<int> AccessTheWebAsync()
{
    // You need to add a reference to System.Net.Http to declare client.
    var client = new HttpClient();

    // GetStringAsync returns a Task<string>. That means that when you await the
    // task you'll get a string (urlContents).
    Task<string> getStringTask = client.GetStringAsync("https://docs.microsoft.com/dotnet");

    // You can do work here that doesn't rely on the string from GetStringAsync.
    DoIndependentWork();

    // The await operator suspends AccessTheWebAsync.
    //  - AccessTheWebAsync can't continue until getStringTask is complete.
    //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    //  - Control resumes here when getStringTask is complete.
    //  - The await operator then retrieves the string result from getStringTask.
    string urlContents = await getStringTask;

    // The return statement specifies an integer result.
    // Any methods that are awaiting AccessTheWebAsync retrieve the length value.
    return urlContents.Length;
}
```

Del ejemplo anterior puede obtener información sobre varios procedimientos. Comience con la firma del método. Incluye el modificador `async`. El tipo de valor devuelto es `Task<int>` (vea la sección "Tipos de valor devuelto" para obtener más opciones). El nombre del método termina en `Async`. En el cuerpo del método, `GetStringAsync` devuelve un elemento `Task<string>`. Esto significa que, cuando se aplica `await` a la tarea, se obtiene un elemento `string` (`urlContents`). Antes de la espera de la tarea, puede realizar otras acciones que no se basen en el elemento `string` de `GetStringAsync`.

Preste mucha atención al operador `await`. Suspende a `AccessTheWebAsync`;

- `AccessTheWebAsync` no puede continuar hasta que se complete `getStringTask`.
- Mientras tanto, el control vuelve al autor de la llamada de `AccessTheWebAsync`.
- Aquí se reanuda el control cuando se completa `getStringTask`.
- Después, el operador `await` recupera el resultado `string` de `getStringTask`.

La instrucción return especifica un resultado entero. Los métodos que están a la espera de `AccessTheWebAsync` recuperan el valor de longitud.

Si `AccessTheWebAsync` no hay ningún trabajo que se pueda hacer entre llamar a `GetStringAsync` y esperar a su finalización, se puede simplificar el código llamando y esperando en la siguiente instrucción única.

```
string urlContents = await client.GetStringAsync("https://docs.microsoft.com/dotnet");
```

Las siguientes características resumen lo que hace que el ejemplo anterior sea un método asincrónico:

- Method Signature incluye un modificador `async`.
- El nombre de un método asincrónico, por convención, finaliza con un sufijo "Async".
- El tipo de valor devuelto es uno de los tipos siguientes:
  - Task<TResult> si el método tiene una instrucción return en la que el operando tiene el tipo `TResult`.
  - Task si el método no tiene ninguna instrucción return ni tiene una instrucción return sin operando.
  - `void` si está escribiendo un controlador de eventos asincrónicos.
  - Cualquier otro tipo que tenga un método `GetAwaiter` (a partir de C# 7.0).

  Para obtener más información, consulte la sección Tipos de valor devuelto y parámetros.
- El método normalmente incluye al menos una expresión `await`, que marca un punto en el que el método no puede continuar hasta que se completa la operación asincrónica en espera. Mientras tanto, se suspende el método y el control vuelve al llamador del método. La sección siguiente de este tema muestra lo que sucede en el punto de suspensión.

En métodos asincrónicos, se utilizan las palabras clave y los tipos proporcionados para indicar lo que se desea hacer y el compilador realiza el resto, incluido el seguimiento de qué debe ocurrir cuando el control vuelve a un punto de espera en un método suspendido. Algunos procesos de rutina, tales como bucles y control de excepciones, pueden ser difíciles de controlar en código asincrónico tradicional. En un método asincrónico, se pueden escribir estos elementos como se haría en una solución sincrónica y se resuelve este problema.

Para obtener más información sobre la asincronía en versiones anteriores de .NET Framework, vea TPL y la programación asincrónica tradicional de .NET Framework.

Lo que ocurre en un método asincrónico

Lo más importante de entender en la programación asincrónica es cómo el flujo de control pasa de un método a otro. El diagrama siguiente lo guía en el proceso:

Los números del diagrama se corresponden a los pasos siguientes, que se inician cuando el usuario hace clic en el botón de inicio.

1. Un controlador de eventos llama al método asincrónico `AccessTheWebAsync` y lo espera.

2. `AccessTheWebAsync` crea una instancia HttpClient y llama al método asincrónico GetStringAsync para descargar el contenido de un sitio web como una cadena.

3. Sucede algo en `GetStringAsync` que suspende el progreso. Quizás debe esperar a un sitio web para realizar la descarga o alguna otra actividad de bloqueo. Para evitar el bloqueo de recursos, `GetStringAsync` genera un control a su llamador, `AccessTheWebAsync`.

   `GetStringAsync` devuelve un elemento Task<TResult>, donde `TResult` es una cadena y `AccessTheWebAsync` asigna la tarea a la variable `getStringTask`. La tarea representa el proceso actual para la llamada a `GetStringAsync`, con el compromiso de generar un valor de cadena real cuando se completa el trabajo.

4. Debido a que `getStringTask` no se ha esperado, `AccessTheWebAsync` puede continuar con otro trabajo que no dependa del resultado final de `GetStringAsync`. Ese trabajo se representa mediante una llamada al método sincrónico `DoIndependentWork`.

5. `DoIndependentWork` es un método sincrónico que funciona y vuelve al llamador.

6. `AccessTheWebAsync` se ha quedado sin el trabajo que se puede realizar sin un resultado de `getStringTask`. Después, `AccessTheWebAsync` desea calcular y devolver la longitud de la cadena descargada, pero el método no puede calcular ese valor hasta que el método tiene la cadena.

   Por consiguiente, `AccessTheWebAsync` utiliza un operador await para suspender el progreso y producir el control al método que llamó `AccessTheWebAsync`. `AccessTheWebAsync` devuelve `Task<int>` al llamador. La tarea representa una sugerencia para generar un resultado entero que es la longitud de la cadena descargada.

> **NOTE**
> Si `GetStringAsync` (y, por tanto, `getStringTask`) se completa antes de que `AccessTheWebAsync` lo espere, el control permanece en `AccessTheWebAsync`. El gasto de suspensión y después regresar a `AccessTheWebAsync` se desperdiciaría si el proceso denominado asincrónico (`getStringTask`) ya se ha completado y `AccessTheWebAsync` no tiene que esperar el resultado final.

Dentro del llamador (el controlador de eventos en este ejemplo), el patrón de procesamiento continúa. El llamador puede hacer otro trabajo que no depende del resultado de `AccessTheWebAsync` antes de esperar ese resultado, o es posible que el llamador se espere inmediatamente. El controlador de eventos está esperando `AccessTheWebAsync`, y `AccessTheWebAsync` está esperando `GetStringAsync`.

7. `GetStringAsync` completa y genera un resultado de la cadena. La llamada a `GetStringAsync` no devuelve el resultado de la cadena de la manera que cabría esperar. (Recuerde que el método ya devolvió una tarea en el paso 3). En su lugar, el resultado de la cadena se almacena en la tarea que representa la finalización del método, `getStringTask`. El operador await recupera el resultado de `getStringTask`. La instrucción de asignación asigna el resultado recuperado a `urlContents`.

8. Cuando `AccessTheWebAsync` tiene el resultado de la cadena, el método puede calcular la longitud de la cadena. El trabajo de `AccessTheWebAsync` también se completa y el controlador de eventos que espera se puede reanudar. En el ejemplo completo del final de este tema, puede comprobar que el controlador de eventos recupera e imprime el valor de resultado de longitud. Si no está familiarizado con la programación asincrónica, reserve un minuto para ver la diferencia entre el comportamiento sincrónico y asincrónico. Un método sincrónico devuelve cuando se completa su trabajo (paso 5), pero un método asincrónico devuelve un valor de tarea cuando se suspende el trabajo (pasos 3 y 6). Cuando el método asincrónico completa finalmente el trabajo, se marca la tarea como completa y el resultado, si existe, se almacena en la tarea.

Para obtener más información sobre el flujo de control, vea Controlar el flujo en los programas asincrónicos (C#).

Métodos asincrónicos de API

Tal vez se pregunte dónde encontrar métodos como `GetStringAsync` que sean compatibles con la programación asincrónica. .NET Framework 4.5 o versiones posteriores y .NET Core contienen muchos miembros que trabajan con `async` y `await`. Puede reconocerlos por el sufijo "Async" que se anexa al nombre del miembro y por su tipo de valor devuelto Task o Task<TResult>. Por ejemplo, la clase `System.IO.Stream` contiene métodos como CopyToAsync, ReadAsync y WriteAsync junto con los métodos sincrónicos CopyTo, Read y Write.

Windows Runtime también contiene muchos métodos que puede utilizar con `async` y `await` en Aplicaciones Windows. Para más información, consulte Subprocesamiento y programación asincrónica para el desarrollo con UWP, y Programación asincrónica (aplicaciones de la Tienda Windows) y Guía de inicio rápido: Llamadas a API asincrónicas en C# o Visual Basic si usa versiones anteriores de Windows Runtime.

Subprocesos

La intención de los métodos Async es ser aplicaciones que no pueden producir bloqueos. Una expresión `await` en un método asincrónico no bloquea el subproceso actual mientras la tarea esperada se encuentra en ejecución. En vez de ello, la expresión declara el resto del método como una continuación y devuelve el control al llamador del método asincrónico.

Las palabras clave `async` y `await` no hacen que se creen subprocesos adicionales. Los métodos Async no requieren multithreading, ya que un método asincrónico no se ejecuta en su propio subproceso. El método se ejecuta en el contexto de sincronización actual y ocupa tiempo en el subproceso únicamente cuando el método está activo. Puede utilizar Task.Run para mover el trabajo enlazado a la CPU a un subproceso en segundo plano,

pero un subproceso en segundo plano no ayuda con un proceso que solo está esperando a que los resultados estén disponibles.

El enfoque basado en asincrónico en la programación asincrónica es preferible a los enfoques existentes en casi todos los casos. En concreto, este enfoque es mejor que la clase BackgroundWorker para las operaciones enlazadas a E/S porque el código es más sencillo y no se tiene que proteger contra las condiciones de carrera. Junto con el método Task.Run, la programación asincrónica es mejor que BackgroundWorker para las operaciones enlazadas a la CPU, ya que la programación asincrónica separa los detalles de coordinación en la ejecución del código del trabajo que `Task.Run` transfiere al grupo de subprocesos.

Async y Await

Si especifica que un método es un método asincrónico mediante el modificador async, habilita las dos funciones siguientes.

- El método asincrónico marcado puede utilizar await para designar puntos de suspensión. El operador `await` indica al compilador que el método asincrónico no puede continuar pasado ese punto hasta que se complete el proceso asincrónico aguardado. Mientras tanto, el control devuelve al llamador del método asincrónico.

  La suspensión de un método asincrónico en una expresión `await` no constituye una salida del método y los bloques `finally` no se ejecutan.

- El método asincrónico marcado sí se puede esperar por los métodos que lo llaman.

Un método asincrónico normalmente contiene una o más apariciones de un operador `await`, pero la ausencia de expresiones `await` no causa errores de compilación. Si un método asincrónico no usa un operador `await` para marcar el punto de suspensión, se ejecuta como un método sincrónico, a pesar del modificador `async`. El compilador detecta una advertencia para dichos métodos.

`async` y `await` son palabras clave contextuales. Para mayor información y ejemplos, vea los siguientes temas:

- async
- await

Tipos de valor devuelto y parámetros

Un método asincrónico suele devolver Task o Task<TResult>. Dentro de un método asincrónico, se aplica un operador `await` a una tarea que devuelve una llamada a otro método asincrónico.

Puede especificar Task<TResult> como el tipo de valor devuelto si el método contiene una instrucción `return` en la que se especifica un operando de tipo `TResult`.

Puede usar Task como tipo de valor devuelto si el método no tiene instrucción return o si tiene una instrucción return que no devuelve un operando.

A partir de C# 7.0, también puede especificar cualquier otro tipo de valor devuelto, siempre que dicho tipo incluya un método `GetAwaiter`. Un ejemplo de este tipo es ValueTask<TResult>. Está disponible en el paquete NuGet System.Threading.Tasks.Extension.

En el ejemplo siguiente se muestra cómo declarar y llamar a un método que devuelve Task<TResult> o Task:

```
// Signature specifies Task<TResult>
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);
    // Return statement specifies an integer result.
    return hours;
}

// Calls to GetTaskOfTResultAsync
Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// or, in a single statement
int intResult = await GetTaskOfTResultAsync();

// Signature specifies Task
async Task GetTaskAsync()
{
    await Task.Delay(0);
    // The method has no return statement.
}

// Calls to GetTaskAsync
Task returnedTask = GetTaskAsync();
await returnedTask;
// or, in a single statement
await GetTaskAsync();
```

Cada tarea devuelta representa el trabajo en curso. Una tarea encapsula la información sobre el estado del proceso asincrónico y, finalmente, el resultado final del proceso o la excepción que el proceso provoca si no tiene éxito.

Un método asincrónico también puede tener un tipo de valor devuelto `void`. Este tipo de valor devuelto se utiliza principalmente para definir controladores de eventos, donde se requiere un tipo de valor devuelto `void`. Los controladores de eventos asincrónicos sirven a menudo como punto de partida para programas asincrónicos.

No se puede esperar a un método asincrónico que tenga un tipo de valor devuelto `void` y el llamador de un método con tipo de valor devuelto void no puede capturar ninguna excepción producida por este.

Un método asincrónico no puede declarar ningún parámetro in, ref o out, pero el método puede llamar a los métodos que tienen estos parámetros. De forma similar, un método asincrónico no puede devolver un valor por referencia, aunque puede llamar a métodos con valores devueltos ref.

Para más información y ejemplos, vea Async Return Types (C#) (Tipos de valor devuelto asincrónicos [C#]). Para más información sobre cómo capturar excepciones en métodos asincrónicos, vea try-catch.

Las API asincrónicas en la programación de Windows Runtime tienen uno de los siguientes tipos de valor devuelto, que son similares a las tareas:

- IAsyncOperation<TResult>, lo que equivale a Task<TResult>
- IAsyncAction, lo que equivale a Task
- IAsyncActionWithProgress<TProgress>
- IAsyncOperationWithProgress<TResult,TProgress>

Convención de nomenclatura

Por convención, los nombres de los métodos que devuelven tipos que suelen admitir "await" (por ejemplo, `Task`, `Task<T>`, `ValueTask` y `ValueTask<T>`) deben terminar por "Async". Los nombres de los métodos que inician operaciones asincrónicas, pero que no devuelven un tipo que admite "await", no deben terminar en "Async", pero pueden empezar por "Begin", "Start" o cualquier otro verbo para sugerir que este método no devuelve ni genera el resultado de la operación.

Puede ignorar esta convención cuando un evento, clase base o contrato de interfaz sugieren un nombre diferente. Por ejemplo, no se debería cambiar el nombre de los controladores de eventos, tales como `Button1_Click`.

Temas relacionados y ejemplos (Visual Studio)

TITLE	DESCRIPCIÓN	EJEMPLO
Tutorial: Acceso a web usando Async y Await (C#)	Muestra cómo convertir una solución WPF sincrónica en una solución WPF asincrónica. La aplicación descarga una serie de sitios web.	Ejemplo de async: Accessing the Web Walkthrough (Ejemplo de async: Tutorial de acceso a web)
Procedimiento para ampliar el tutorial de async usando Task.WhenAll (C#)	Agrega Task.WhenAll al tutorial anterior. El uso de `WhenAll` inicia todas las descargas al mismo tiempo.	
Procedimiento para realizar varias solicitudes web en paralelo con async y await (C#)	Demuestra cómo comenzar varias tareas al mismo tiempo.	Ejemplo de async: Make Multiple Web Requests in Parallel (Ejemplo de async: Realizar varias solicitudes web en paralelo)
Tipos de valor devueltos asincrónicos (C#)	Muestra los tipos que los métodos asincrónicos pueden devolver y explica cuándo es apropiado cada uno de ellos.	
Controlar el flujo en los programas asincrónicos (C#)	Rastrea en detalle el flujo de control a través de una sucesión de expresiones await en un programa asincrónico.	Ejemplo de async: Control Flow in Async Programs (Ejemplo de async: Flujo de control en programas de async)
Ajustar una aplicación asincrónica (C#)	Muestra cómo agregar la siguiente funcionalidad a la solución asincrónica: - Cancel an Async Task or a List of Tasks (C#) (Cancelación de una tarea asincrónica o de una lista de tareas [C#]) - Cancel Async Tasks after a Period of Time (C#) (Cancelación de tareas asincrónicas tras un período de tiempo [C#]) - Cancel Remaining Async Tasks after One Is Complete (C#) (Cancelación de tareas asincrónicas restantes cuando se completa una [C#]) - Start Multiple Async Tasks and Process Them As They Complete (C#) (Inicio de varias tareas asincrónicas y cómo procesarlas a medida que se completan [C#])	Ejemplo de async: Fine Tuning Your Application (Ejemplo de async: Ajuste de la aplicación)
Controlar la reentrada en aplicaciones asincrónicas (C#)	Muestra cómo controlar los casos en los que se reinicia una operación asincrónica activa mientras se ejecuta.	

TITLE	DESCRIPCIÓN	EJEMPLO
WhenAny: Bridging between the .NET Framework and the Windows Runtime (Puente entre .NET Framework y Windows Runtime)	Muestra cómo unir entre tipos de tareas en .NET Framework e IAsyncOperations en Windows Runtime para poder usar WhenAny con un método de Windows Runtime.	Ejemplo de async: Bridging between .NET and Windows Runtime (AsTask and WhenAny) (Ejemplo de async: Puente entre .NET y Windows Runtime [AsTask y WhenAny])
Async Cancellation: Bridging between the .NET Framework and the Windows Runtime (Puente entre NET. Framework y Windows Runtime)	Muestra cómo unir entre tipos de tareas en .NET Framework e IAsyncOperations en Windows Runtime para poder usar CancellationTokenSource con un método de Windows Runtime.	Ejemplo de async: Bridging between .NET and Windows Runtime (AsTask & Cancellation) (Ejemplo de async: Puente entre .NET y Windows Runtime [AsTask y Cancellation])
Usar Async en acceso a archivos (C#)	Enumera y demuestra los beneficios de usar async y await para obtener acceso a archivos.	
Modelo asincrónico basado en tareas (TAP)	Describe un nuevo patrón de asincronía en .NET Framework. El patrón está basado en los tipos Task y Task<TResult>.	
Vídeos de Async en Channel 9	Proporciona vínculos a una serie de vídeos sobre programación asincrónica.	

Ejemplo completo

El código siguiente es el archivo *MainWindow.xaml.cs* de la aplicación de WPF que se explica en este artículo. Puede descargar el ejemplo de Async Sample: Example from "Asynchronous Programming with Async and Await" (Ejemplo de async: Ejemplo de "Programación asincrónica con Async y Await).

```
using System;
using System.Threading.Tasks;
using System.Windows;

// Add a using directive and a reference for System.Net.Http;
using System.Net.Http;

namespace AsyncFirstExample
{
    public partial class MainWindow : Window
    {
        // Mark the event handler with async so you can use await in it.
        private async void StartButton_Click(object sender, RoutedEventArgs e)
        {
            // Call and await separately.
            //Task<int> getLengthTask = AccessTheWebAsync();
            //// You can do independent work here.
            //int contentLength = await getLengthTask;

            int contentLength = await AccessTheWebAsync();

            resultsTextBox.Text +=
                $"\r\nLength of the downloaded string: {contentLength}.\r\n";
        }

        // Three things to note in the signature:
        //  - The method has an async modifier.
        //  - The return type is Task or Task<T>. (See "Return Types" section.)
        //    Here, it is Task<int> because the return statement returns an integer.
        //  - The method name ends in "Async."
        async Task<int> AccessTheWebAsync()
        {
            // You need to add a reference to System.Net.Http to declare client.
            var client = new HttpClient();

            // GetStringAsync returns a Task<string>. That means that when you await the
            // task you'll get a string (urlContents).
            Task<string> getStringTask = client.GetStringAsync("https://docs.microsoft.com/dotnet");

            // You can do work here that doesn't rely on the string from GetStringAsync.
            DoIndependentWork();

            // The await operator suspends AccessTheWebAsync.
            //  - AccessTheWebAsync can't continue until getStringTask is complete.
            //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
            //  - Control resumes here when getStringTask is complete.
            //  - The await operator then retrieves the string result from getStringTask.
            string urlContents = await getStringTask;

            // The return statement specifies an integer result.
            // Any methods that are awaiting AccessTheWebAsync retrieve the length value.
            return urlContents.Length;
        }

        void DoIndependentWork()
        {
            resultsTextBox.Text += "\r\nWorking . . . . . .\r\n";
        }
    }
}

// Sample Output:

// Working . . . . . .

// Length of the downloaded string: 41564.
```

Vea también

- async
- await
- Programación asincrónica
- Información general de Async

Tutorial: Acceso a web usando Async y Await (C#)

29/06/2020 • 28 minutes to read • Edit Online

Puede escribir programas asincrónicos de manera más fácil e intuitiva usando las características async/await. Puede escribir código asincrónico parecido al código sincrónico y dejar que el compilador gestione las difíciles funciones de devolución de llamada y continuaciones que normalmente implica el código asincrónico.

Para obtener más información sobre la característica Async, consulte Programación asincrónica con async y await (C#).

Este tutorial comienza con una aplicación sincrónica de Windows Presentation Foundation (WPF) que suma el número de bytes de una lista de sitios web. A continuación, el tutorial convierte la aplicación en una solución asincrónica mediante el uso de las características nuevas.

Si no quiere compilar usted mismo las aplicaciones, puede descargar Ejemplo de Async: obtener acceso al tutorial web (C# y Visual Basic).

> **NOTE**
> Para ejecutar los ejemplos, debe tener Visual Studio 2012 o posterior, y .NET Framework 4.5 o posterior, instalado en el equipo.

Crear una aplicación WPF

1. Inicie Visual Studio.
2. En la barra de menús, elija **Archivo** > **Nuevo** > **Proyecto**.

   Aparece el cuadro de diálogo **Nuevo proyecto** .
3. En el panel **Plantillas instaladas**, elija Visual C# y después elija **Aplicación WPF** en la lista de tipos de proyecto.
4. En el cuadro de texto **Nombre**, escriba `AsyncExampleWPF` y elija el botón **Aceptar**.

   El proyecto nuevo aparece en el **Explorador de soluciones**.

Diseñar una ventana MainWindow simple de WPF

1. En el Editor de código de Visual Studio, elija la pestaña **MainWindow.xaml** .
2. Si la ventana **Cuadro de herramientas** no se está mostrando, abra el menú **Vista** y elija **Cuadro de herramientas**.
3. Agregue un control **Botón** y un control **TextBox** a la ventana **MainWindow**.
4. Resalte el control **TextBox** y, en la ventana **Propiedades**, establezca los siguientes valores:
   - Establezca la propiedad **Nombre** en `resultsTextBox`.
   - Establezca la propiedad **Alto** en 250.
   - Establezca la propiedad **Ancho** en 500.
   - En la pestaña **Texto**, especifique una fuente monoespaciada, como Lucida Console o Global

Monospace.

5. Resalte el control **Botón** y, en la ventana **Propiedades**, establezca los siguientes valores:
   - Establezca la propiedad **Nombre** en `startButton`.
   - Cambie el valor de la propiedad **Contenido** de **Botón** a **Inicio**.
6. Coloque el cuadro de texto y el botón de manera que ambos aparezcan en la ventana **MainWindow**.

   Para obtener más información sobre el Diseñador XAML de WPF, consulte Crear una IU con el Diseñador XAML.

Agregar una referencia

1. En el **Explorador de soluciones**, resalte el nombre del proyecto.
2. En la barra de menús, elija **Proyecto** > **Agregar referencia**.

   Aparecerá el cuadro de diálogo **Administrador de referencias**.
3. En la parte superior del cuadro de diálogo, compruebe que el proyecto tiene como destino .NET Framework 4.5 o superior.
4. En la categoría **Ensamblados**, elija **Framework** si no está ya seleccionado.
5. En la lista de nombres, seleccione la casilla **System.Net.Http**.
6. Elija el botón **Aceptar** para cerrar el cuadro de diálogo.

Agregar las directivas using necesarias

1. En el **Explorador de soluciones**, abra el menú contextual de MainWindow.xaml.cs y después elija **Ver código**.
2. Agregue las siguientes directivas `using` en la parte superior del archivo de código si no están ya presentes.

```
using System.Net.Http;
using System.Net;
using System.IO;
```

Crear una aplicación sincrónica

1. En la ventana de diseño, MainWindow.xaml, haga doble clic en el botón **Inicio** para crear el controlador de eventos `startButton_Click` en MainWindow.xaml.cs.
2. En MainWindow.xaml.cs, copie el código siguiente en el cuerpo de `startButton_Click`:

```
resultsTextBox.Clear();
SumPageSizes();
resultsTextBox.Text += "\r\nControl returned to startButton_Click.";
```

   El código llama al método que controla la aplicación, `SumPageSizes`, y muestra un mensaje cuando el control vuelve a `startButton_Click`.
3. El código de la solución sincrónica contiene los cuatro métodos siguientes:

- `SumPageSizes`, que obtiene una lista de direcciones URL de páginas web de `SetUpURLList` y, a continuación, llama a `GetURLContents` y `DisplayResults` para que procesen cada dirección URL.
- `SetUpURLList`, que crea y devuelve una lista de direcciones web.
- `GetURLContents`, que descarga el contenido de cada sitio web y devuelve el contenido como una matriz de bytes.
- `DisplayResults`, que muestra el número de bytes de la matriz de bytes de cada dirección URL.

Copie los cuatro métodos siguientes y péguelos bajo el controlador de eventos `startButton_Click` en MainWindow.xaml.cs:

```
private void SumPageSizes()
{
    // Make a list of web addresses.
    List<string> urlList = SetUpURLList();

    var total = 0;
    foreach (var url in urlList)
    {
        // GetURLContents returns the contents of url as a byte array.
        byte[] urlContents = GetURLContents(url);

        DisplayResults(url, urlContents);

        // Update the total.
        total += urlContents.Length;
    }

    // Display the total count for all of the web addresses.
    resultsTextBox.Text += $"\r\n\r\nTotal bytes returned:  {total}\r\n";
}

private List<string> SetUpURLList()
{
    var urls = new List<string>
    {
        "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
        "https://msdn.microsoft.com",
        "https://msdn.microsoft.com/library/hh290136.aspx",
        "https://msdn.microsoft.com/library/ee256749.aspx",
        "https://msdn.microsoft.com/library/hh290138.aspx",
        "https://msdn.microsoft.com/library/hh290140.aspx",
        "https://msdn.microsoft.com/library/dd470362.aspx",
        "https://msdn.microsoft.com/library/aa578028.aspx",
        "https://msdn.microsoft.com/library/ms404677.aspx",
        "https://msdn.microsoft.com/library/ff730837.aspx"
    };
    return urls;
}

private byte[] GetURLContents(string url)
{
    // The downloaded resource ends up in the variable named content.
    var content = new MemoryStream();

    // Initialize an HttpWebRequest for the current URL.
    var webReq = (HttpWebRequest)WebRequest.Create(url);

    // Send the request to the Internet resource and wait for
    // the response.
    // Note: you can't use HttpWebRequest.GetResponse in a Windows Store app.
    using (WebResponse response = webReq.GetResponse())
    {
        // Get the data stream that is associated with the specified URL.
```

```
        using (Stream responseStream = response.GetResponseStream())
        {
            // Read the bytes in responseStream and copy them to content.
            responseStream.CopyTo(content);
        }
    }

    // Return the result as a byte array.
    return content.ToArray();
}

private void DisplayResults(string url, byte[] content)
{
    // Display the length of each website. The string format
    // is designed to be used with a monospaced font, such as
    // Lucida Console or Global Monospace.
    var bytes = content.Length;
    // Strip off the "https://".
    var displayURL = url.Replace("https://", "");
    resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
}
```

Probar la solución sincrónica

Presione la tecla **F5** para ejecutar el programa y elija el botón **Inicio**.

Debería aparecer un resultado similar a la lista siguiente:

```
msdn.microsoft.com/library/windows/apps/br211380.aspx        383832
msdn.microsoft.com                                            33964
msdn.microsoft.com/library/hh290136.aspx               225793
msdn.microsoft.com/library/ee256749.aspx               143577
msdn.microsoft.com/library/hh290138.aspx               237372
msdn.microsoft.com/library/hh290140.aspx               128279
msdn.microsoft.com/library/dd470362.aspx               157649
msdn.microsoft.com/library/aa578028.aspx               204457
msdn.microsoft.com/library/ms404677.aspx               176405
msdn.microsoft.com/library/ff730837.aspx               143474

Total bytes returned:  1834802

Control returned to startButton_Click.
```

Tenga en cuenta que los recuentos tardan unos segundos en mostrarse. Durante ese tiempo, el subproceso de interfaz de usuario se bloquea mientras espera a que se descarguen los recursos solicitados. Como resultado, no se puede mover, maximizar, minimizar o ni siquiera cerrar la ventana de la pantalla después de elegir el botón **Inicio**. Estos intentos producirán un error hasta que empiecen a aparecer los recuentos de bytes. Si un sitio web no responde, no se le indicará cuál es el sitio que produjo el error. Incluso resulta difícil dejar de esperar y cerrar el programa.

Convertir GetURLContents en un método asincrónico

1. Para convertir la solución sincrónica a una solución asincrónica, lo mejor es comenzar en `GetURLContents`, ya que las llamadas al método HttpWebRequestGetResponse y al método StreamCopyTo son el lugar desde donde la aplicación accede a la web. .NET Framework facilita la conversión proporcionando versiones asincrónicas de ambos métodos.

   Para obtener más información sobre los modelos que se usan en `GetURLContents`, vea WebRequest.

> **NOTE**
> A medida que siga los pasos de este tutorial, aparecerán varios errores del compilador. Puede hacer caso omiso y continuar con el tutorial.

Cambie el método al que se llama en la tercera línea de `GetURLContents` de `GetResponse` al método GetResponseAsync asincrónico basado en tareas.

```
using (WebResponse response = webReq.GetResponseAsync())
```

2. `GetResponseAsync` devuelve Task<TResult>. En este caso, la *variable de devolución de tarea*, `TResult`, tiene un tipo WebResponse. La tarea es una promesa para generar un objeto `WebResponse` real después de que se descarguen los datos solicitados y la tarea se ejecute hasta completarse.

   Para recuperar el valor `WebResponse` de la tarea, aplique un operador await a la llamada a `GetResponseAsync`, como se muestra en el código siguiente.

```
using (WebResponse response = await webReq.GetResponseAsync())
```

   El operador `await` suspende la ejecución del método actual, `GetURLContents`, hasta que se complete la tarea esperada. Mientras tanto, el control devuelve al llamador del método actual. En este ejemplo, el método actual es `GetURLContents` y el llamador es `SumPageSizes`. Cuando la tarea finaliza, el objeto `WebResponse` prometido se genera como valor de la tarea esperada y se asigna a la variable `response`.

   La instrucción anterior se puede dividir en las dos instrucciones siguientes para aclarar lo que sucede.

```
//Task<WebResponse> responseTask = webReq.GetResponseAsync();
//using (WebResponse response = await responseTask)
```

   La llamada a `webReq.GetResponseAsync` devuelve `Task(Of WebResponse)` o `Task<WebResponse>`. A continuación, se aplica un operador await a la tarea para recuperar el valor `WebResponse`.

   Si el método asincrónico debe realizar un trabajo que no depende de la finalización de la tarea, el método puede continuar con ese trabajo entre estas dos instrucciones, después de la llamada al método asincrónico y antes de que se aplique el operador `await`. Para obtener ejemplos, vea Procedimiento para realizar varias solicitudes web en paralelo con async y await (C#) y Procedimiento para ampliar el tutorial de async usando Task.WhenAll (C#).

3. Dado que ha agregado el operador `await` en el paso anterior, se produce un error del compilador. El operador puede usarse únicamente en los métodos marcados con el modificador async. Omita el error mientras repita los pasos de conversión para reemplazar la llamada a `CopyTo` con una llamada a `CopyToAsync`.

   - Cambie el nombre del método al que se llama a CopyToAsync.
   - El método `CopyTo` o `CopyToAsync` copia bytes a su argumento, `content`, y no devuelve un valor significativo. En la versión sincrónica, la llamada a `CopyTo` es una sencilla instrucción que no devuelve un valor. La versión asincrónica, `CopyToAsync`, devuelve una Task. La tarea funciona como "Task(void)" y permite que se espere al método. Aplique `Await` o `await` a la llamada a `CopyToAsync`, como se muestra en el código siguiente.

```
await responseStream.CopyToAsync(content);
```

La instrucción anterior abrevia las dos líneas de código siguientes.

```
// CopyToAsync returns a Task, not a Task<T>.
//Task copyTask = responseStream.CopyToAsync(content);

// When copyTask is completed, content contains a copy of
// responseStream.
//await copyTask;
```

4. Lo único que queda por hacer en `GetURLContents` es ajustar la firma del método. Puede usar el operador `await` únicamente en los métodos marcados con el modificador async. Agregue el modificador para marcar el método como *método asincrónico*, como se muestra en el código siguiente.

```
private async byte[] GetURLContents(string url)
```

5. El tipo de valor devuelto de un método asincrónico solo puede ser Task, Task<TResult> o `void` en C#. Normalmente, el tipo de valor devuelto `void` se usa solo en un controlador de eventos asincrónico, en el que se requiere `void`. En otros casos, se usa `Task(T)` si el método completado tiene una instrucción return que devuelve un valor de tipo T, y se usa `Task` si el método completado no devuelve un valor significativo. Puede considerar que el tipo de valor devuelto `Task` significa "Task(void)".

   Para obtener más información, consulte Tipos de valor devueltos asincrónicos (C#).

   El método `GetURLContents` tiene una instrucción return, y la instrucción devuelve una matriz de bytes. Por lo tanto, el tipo de valor devuelto de la versión de async es Task(T), donde T es una matriz de bytes. Realice los cambios siguientes en la firma del método:

   - Cambie el tipo de valor devuelto a `Task<byte[]>`.
   - Por convención, los métodos asincrónicos tienen nombres que terminan en "Async", por lo que debe cambiar el nombre del método a `GetURLContentsAsync`.

   En el código siguiente se muestran estos cambios.

```
private async Task<byte[]> GetURLContentsAsync(string url)
```

   Con estos pocos cambios, se completa la conversión de `GetURLContents` en un método asincrónico.

Convertir SumPageSizes en un método asincrónico

1. Repita los pasos del procedimiento anterior para `SumPageSizes`. Primero, cambie la llamada a `GetURLContents` a una llamada asincrónica.

   - Cambie el nombre del método al que se llama de `GetURLContents` a `GetURLContentsAsync`, si aún no lo ha hecho.
   - Aplique `await` a la tarea que devuelve `GetURLContentsAsync` para obtener el valor de la matriz de bytes.

   En el código siguiente se muestran estos cambios.

```
byte[] urlContents = await GetURLContentsAsync(url);
```

La asignación anterior abrevia las dos líneas de código siguientes.

```
// GetURLContentsAsync returns a Task<T>. At completion, the task
// produces a byte array.
//Task<byte[]> getContentsTask = GetURLContentsAsync(url);
//byte[] urlContents = await getContentsTask;
```

2. Realice los cambios siguientes en la firma del método:

   - Marque el método con el modificador `async`.
   - Agregue "Async" al nombre del método.
   - Esta vez no hay ninguna variable de devolución de tarea, T, porque `SumPageSizesAsync` no devuelve un valor para T. (El método no tiene ninguna instrucción `return`). Sin embargo, el método debe devolver `Task` para admitir await. Por tanto, cambie el tipo de valor devuelto del método de `void` a `Task`.

   En el código siguiente se muestran estos cambios.

```
private async Task SumPageSizesAsync()
```

   Se completa así la conversión de `SumPageSizes` a `SumPageSizesAsync`.

Convertir startButton_Click en un método asincrónico

1. En el controlador de eventos, cambie el nombre del método llamado de `SumPageSizes` a `SumPageSizesAsync`, si aún no lo ha hecho.

2. Dado que `SumPageSizesAsync` es un método asincrónico, cambie el código en el controlador de eventos para esperar el resultado.

   La llamada a `SumPageSizesAsync` refleja la llamada a `CopyToAsync` en `GetURLContentsAsync`. La llamada devuelve `Task`, no `Task(T)`.

   Al igual que en los procedimientos anteriores, puede convertir la llamada usando una o dos instrucciones. En el código siguiente se muestran estos cambios.

```
// One-step async call.
await SumPageSizesAsync();

// Two-step async call.
//Task sumTask = SumPageSizesAsync();
//await sumTask;
```

3. Para evitar volver a entrar accidentalmente en la operación, agregue la instrucción siguiente encima de `startButton_Click` para deshabilitar el botón **Inicio**.

```
// Disable the button until the operation is complete.
startButton.IsEnabled = false;
```

   Puede volver a habilitar el botón al final del controlador de eventos.

```
// Reenable the button in case you want to run the operation again.
startButton.IsEnabled = true;
```

Para obtener más información sobre la reentrada, consulte Handling Reentrancy in Async Apps (C#) (Controlar la reentrada en aplicaciones asincrónicas [C#]).

4. Finalmente, agregue el modificador `async` a la declaración de modo que el controlador de eventos pueda esperar `SumPagSizesAsync`.

```
private async void startButton_Click(object sender, RoutedEventArgs e)
```

Normalmente, no se cambian los nombres de los controladores de eventos. El tipo de valor devuelto no se cambia a `Task` porque los controladores de eventos deben devolver `void`.

Así se completa la conversión del proyecto de procesamiento sincrónico a asincrónico.

Probar la solución asincrónica

1. Presione la tecla **F5** para ejecutar el programa y elija el botón **Inicio**.

2. Deberían aparecer resultados similares a los de la solución sincrónica. No obstante, debe tener en cuenta las siguientes diferencias.

   - No todos los resultados se producen al mismo tiempo cuando se completa el procesamiento. Por ejemplo, ambos programas contienen una línea en `startButton_Click` que borra el cuadro de texto. La intención es borrar el cuadro de texto entre ejecuciones si elige el botón **Inicio** por segunda vez, después de que haya aparecido un conjunto de resultados. En la versión sincrónica, el cuadro de texto se borra antes de que aparezcan los recuentos por segunda vez, cuando se completen las descargas y el subproceso de interfaz de usuario esté libre para llevar a cabo otro trabajo. En la versión asincrónica, el cuadro de texto se borra inmediatamente después de que se elija el botón **Inicio**.

   - Lo más importante es que el subproceso de interfaz de usuario no se bloquea durante las descargas. Puede mover o cambiar el tamaño de la ventana mientras se descargan, se cuentan y se muestran los recursos web. Si uno de los sitios web es lento o no responde, puede cancelar la operación eligiendo el botón **Cerrar** (la X en el campo de color rojo en la esquina superior derecha).

Reemplazar el método GetURLContentsAsync con un método de .NET

1. .NET Framework 4.5 y las versiones posteriores proporcionan muchos métodos asincrónicos que puede usar. Uno de ellos, el método HttpClientGetByteArrayAsync(String), hace justo lo que necesita para este tutorial. Se puede usar en lugar del método `GetURLContentsAsync` que creó en un procedimiento anterior.

   El primer paso es crear un objeto `HttpClient` en el método `SumPageSizesAsync`. Agregue la siguiente declaración al principio del método.

```
// Declare an HttpClient object and increase the buffer size. The
// default buffer size is 65,536.
HttpClient client =
    new HttpClient() { MaxResponseContentBufferSize = 1000000 };
```

2. En `SumPageSizesAsync,` reemplace la llamada a su método `GetURLContentsAsync` con una llamada al

método `HttpClient`.

```
byte[] urlContents = await client.GetByteArrayAsync(url);
```

3. Quite o marque como comentario el método `GetURLContentsAsync` que escribió.

4. Presione la tecla **F5** para ejecutar el programa y elija el botón **Inicio**.

   El comportamiento de esta versión del proyecto debería coincidir con el comportamiento que se describe en el procedimiento "Para probar la solución asincrónica", pero con incluso menos trabajo por su parte.

código de ejemplo

El código siguiente contiene el ejemplo completo de la conversión de una solución sincrónica a una asincrónica usando el método `GetURLContentsAsync` asincrónico que escribió. Observe que es muy parecida a la solución sincrónica original.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add the following using directives, and add a reference for System.Net.Http.
using System.Net.Http;
using System.IO;
using System.Net;

namespace AsyncExampleWPF
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // Disable the button until the operation is complete.
            startButton.IsEnabled = false;

            resultsTextBox.Clear();

            // One-step async call.
            await SumPageSizesAsync();

            // Two-step async call.
            //Task sumTask = SumPageSizesAsync();
            //await sumTask;

            resultsTextBox.Text += "\r\nControl returned to startButton_Click.\r\n";

            // Reenable the button in case you want to run the operation again.
```

```
        // Reenable the button in case you want to run the operation again.
        startButton.IsEnabled = true;
    }

    private async Task SumPageSizesAsync()
    {
        // Make a list of web addresses.
        List<string> urlList = SetUpURLList();

        var total = 0;

        foreach (var url in urlList)
        {
            byte[] urlContents = await GetURLContentsAsync(url);

            // The previous line abbreviates the following two assignment statements.

            // GetURLContentsAsync returns a Task<T>. At completion, the task
            // produces a byte array.
            //Task<byte[]> getContentsTask = GetURLContentsAsync(url);
            //byte[] urlContents = await getContentsTask;

            DisplayResults(url, urlContents);

            // Update the total.
            total += urlContents.Length;
        }
        // Display the total count for all of the websites.
        resultsTextBox.Text +=
            $"\r\n\r\nTotal bytes returned:  {total}\r\n";
    }

    private List<string> SetUpURLList()
    {
        List<string> urls = new List<string>
        {
            "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
            "https://msdn.microsoft.com",
            "https://msdn.microsoft.com/library/hh290136.aspx",
            "https://msdn.microsoft.com/library/ee256749.aspx",
            "https://msdn.microsoft.com/library/hh290138.aspx",
            "https://msdn.microsoft.com/library/hh290140.aspx",
            "https://msdn.microsoft.com/library/dd470362.aspx",
            "https://msdn.microsoft.com/library/aa578028.aspx",
            "https://msdn.microsoft.com/library/ms404677.aspx",
            "https://msdn.microsoft.com/library/ff730837.aspx"
        };
        return urls;
    }

    private async Task<byte[]> GetURLContentsAsync(string url)
    {
        // The downloaded resource ends up in the variable named content.
        var content = new MemoryStream();

        // Initialize an HttpWebRequest for the current URL.
        var webReq = (HttpWebRequest)WebRequest.Create(url);

        // Send the request to the Internet resource and wait for
        // the response.
        using (WebResponse response = await webReq.GetResponseAsync())

        // The previous statement abbreviates the following two statements.

        //Task<WebResponse> responseTask = webReq.GetResponseAsync();
        //using (WebResponse response = await responseTask)
        {
            // Get the data stream that is associated with the specified url.
            using (Stream responseStream = response.GetResponseStream())
            {
```

```
                {
                    // Read the bytes in responseStream and copy them to content.
                    await responseStream.CopyToAsync(content);

                    // The previous statement abbreviates the following two statements.

                    // CopyToAsync returns a Task, not a Task<T>.
                    //Task copyTask = responseStream.CopyToAsync(content);

                    // When copyTask is completed, content contains a copy of
                    // responseStream.
                    //await copyTask;
                }
            }
            // Return the result as a byte array.
            return content.ToArray();
        }

        private void DisplayResults(string url, byte[] content)
        {
            // Display the length of each website. The string format
            // is designed to be used with a monospaced font, such as
            // Lucida Console or Global Monospace.
            var bytes = content.Length;
            // Strip off the "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

El código siguiente contiene el ejemplo completo de la solución que usa el método `HttpClient`, `GetByteArrayAsync`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add the following using directives, and add a reference for System.Net.Http.
using System.Net.Http;
using System.IO;
using System.Net;

namespace AsyncExampleWPF
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            resultsTextBox.Clear();
```

```
        // Disable the button until the operation is complete.
        startButton.IsEnabled = false;

        // One-step async call.
        await SumPageSizesAsync();

        //// Two-step async call.
        //Task sumTask = SumPageSizesAsync();
        //await sumTask;

        resultsTextBox.Text += "\r\nControl returned to startButton_Click.\r\n";

        // Reenable the button in case you want to run the operation again.
        startButton.IsEnabled = true;
    }

    private async Task SumPageSizesAsync()
    {
        // Declare an HttpClient object and increase the buffer size. The
        // default buffer size is 65,536.
        HttpClient client =
            new HttpClient() { MaxResponseContentBufferSize = 1000000 };

        // Make a list of web addresses.
        List<string> urlList = SetUpURLList();

        var total = 0;

        foreach (var url in urlList)
        {
            // GetByteArrayAsync returns a task. At completion, the task
            // produces a byte array.
            byte[] urlContents = await client.GetByteArrayAsync(url);

            // The following two lines can replace the previous assignment statement.
            //Task<byte[]> getContentsTask = client.GetByteArrayAsync(url);
            //byte[] urlContents = await getContentsTask;

            DisplayResults(url, urlContents);

            // Update the total.
            total += urlContents.Length;
        }

        // Display the total count for all of the websites.
        resultsTextBox.Text +=
            $"\r\n\r\nTotal bytes returned:  {total}\r\n";
    }

    private List<string> SetUpURLList()
    {
        List<string> urls = new List<string>
        {
            "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
            "https://msdn.microsoft.com",
            "https://msdn.microsoft.com/library/hh290136.aspx",
            "https://msdn.microsoft.com/library/ee256749.aspx",
            "https://msdn.microsoft.com/library/hh290138.aspx",
            "https://msdn.microsoft.com/library/hh290140.aspx",
            "https://msdn.microsoft.com/library/dd470362.aspx",
            "https://msdn.microsoft.com/library/aa578028.aspx",
            "https://msdn.microsoft.com/library/ms404677.aspx",
            "https://msdn.microsoft.com/library/ff730837.aspx"
        };
        return urls;
    }

    private void DisplayResults(string url, byte[] content)
    {
```

```
            // Display the length of each website. The string format
            // is designed to be used with a monospaced font, such as
            // Lucida Console or Global Monospace.
            var bytes = content.Length;
            // Strip off the "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

Vea también

- Ejemplo de async: Acceso al tutorial web (C# y Visual Basic)
- async
- await
- Programación asincrónica con Async y Await (C#)
- Tipos de valor devueltos asincrónicos (C#)
- Task-based Asynchronous Programming (TAP) (Programación asincrónica basada en tareas [TAP])
- Procedimiento para ampliar el tutorial de async usando Task.WhenAll (C#)
- Procedimiento para realizar varias solicitudes web en paralelo con async y await (C#)

Procedimiento para ampliar el tutorial de async usando Task.WhenAll (C#)

18/03/2020 • 15 minutes to read • Edit Online

El rendimiento de la solución asincrónica de Walkthrough: Accessing the Web by Using async and await (C#) (Tutorial: acceso a la web con async y await [C#]) se puede mejorar con el método Task.WhenAll. Este método espera de forma asincrónica varias operaciones asincrónicas, que se representan como una colección de tareas.

Es posible que en el tutorial observara que los sitios web se descargan a distintas velocidades. A veces uno de los sitios web es muy lento, lo que retrasa las descargas restantes. Cuando ejecute las soluciones asincrónicas que se compilan en el tutorial, puede finalizar el programa fácilmente si no quiere esperar, pero una mejor opción sería iniciar todas las descargas al mismo tiempo y dejar que las más rápidas continúen sin tener que esperar a la que se retrasa.

El método `Task.WhenAll` se aplica a una colección de tareas. La aplicación de `WhenAll` devuelve una única tarea que no está completa hasta que se completen todas las tareas de la colección. La tarea parece ejecutarse en paralelo, pero no se crean subprocesos adicionales. Las tareas se pueden completar en cualquier orden.

> **IMPORTANT**
>
> Los procedimientos siguientes describen las extensiones para las aplicaciones asincrónicas que se desarrollan en Walkthrough: Accessing the Web by Using async and await (C#) (Tutorial: Acceso a la web con async y await [C#]). Puede desarrollar las aplicaciones completando el tutorial o descargando el código de Ejemplos de código para desarrolladores.
>
> Para ejecutar el ejemplo, debe tener instalado Visual Studio 2012 o una versión posterior en el equipo.

Para agregar Task.WhenAll a la solución GetURLContentsAsync

1. Agregue el método `ProcessURLAsync` a la primera aplicación que se desarrolla en Walkthrough: Accessing the Web by Using async and await (C#) (Tutorial: Acceso a la web con async y await [C#]).

   - Si descargó el código de Ejemplos de código para desarrolladores, abra el proyecto AsyncWalkthrough y, después, agregue `ProcessURLAsync` al archivo MainWindow.xaml.cs.
   - Si desarrolló el código completando el tutorial, agregue `ProcessURLAsync` a la aplicación que incluye el método `GetURLContentsAsync`. El archivo MainWindow.xaml.cs para esta aplicación es el primer ejemplo en la sección "Ejemplos de código completados desde el tutorial".

   El método `ProcessURLAsync` consolida las acciones en el cuerpo del bucle `foreach` de `SumPageSizesAsync` en el tutorial original. El método descarga de forma asincrónica el contenido de un sitio web especificado como una matriz de bytes y, después, muestra y devuelve la longitud de la matriz de bytes.

   ```
   private async Task<int> ProcessURLAsync(string url)
   {
       var byteArray = await GetURLContentsAsync(url);
       DisplayResults(url, byteArray);
       return byteArray.Length;
   }
   ```

2. Convierta en comentario o elimine el bucle `foreach` de `SumPageSizesAsync`, como se muestra en el código siguiente.

```
//var total = 0;
//foreach (var url in urlList)
//{
//    byte[] urlContents = await GetURLContentsAsync(url);

//    // The previous line abbreviates the following two assignment statements.
//    // GetURLContentsAsync returns a Task<T>. At completion, the task
//    // produces a byte array.
//    //Task<byte[]> getContentsTask = GetURLContentsAsync(url);
//    //byte[] urlContents = await getContentsTask;

//    DisplayResults(url, urlContents);

//    // Update the total.
//    total += urlContents.Length;
//}
```

3. Cree una colección de tareas. El código siguiente define una consulta que, cuando la ejecute el método ToArray, crea una colección de tareas que descargan el contenido de cada sitio web. Las tareas se inician cuando se evalúa la consulta.

   Agregue el código siguiente en el método `SumPageSizesAsync` después de la declaración de `urlList`.

```
// Create a query.
IEnumerable<Task<int>> downloadTasksQuery =
    from url in urlList select ProcessURLAsync(url);

// Use ToArray to execute the query and start the download tasks.
Task<int>[] downloadTasks = downloadTasksQuery.ToArray();
```

4. Aplique `Task.WhenAll` a la colección de tareas, `downloadTasks`. `Task.WhenAll` devuelve una única tarea que finaliza cuando se completan todas las tareas de la colección de tareas.

   En el ejemplo siguiente, la expresión `await` espera la finalización de la única tarea que devuelve `WhenAll`. La expresión se evalúa como una matriz de enteros, donde cada entero es la longitud de un sitio web descargado. Agregue el código siguiente a `SumPageSizesAsync`, después del código que agregó en el paso anterior.

```
// Await the completion of all the running tasks.
int[] lengths = await Task.WhenAll(downloadTasks);

//// The previous line is equivalent to the following two statements.
//Task<int[]> whenAllTask = Task.WhenAll(downloadTasks);
//int[] lengths = await whenAllTask;
```

5. Por último, use el método Sum para calcular la suma de las longitudes de todos los sitios web. Agregue la línea siguiente a `SumPageSizesAsync`.

```
int total = lengths.Sum();
```

Para agregar Task.WhenAll a la solución HttpClient.GetByteArrayAsync

1. Agregue la siguiente versión de `ProcessURLAsync` a la segunda aplicación que se desarrolla en Walkthrough: Accessing the Web by Using async and await (C#) (Tutorial: Acceso a la web con async y await [C#]).

   - Si descargó el código de Ejemplos de código para desarrolladores, abra el proyecto AsyncWalkthrough_HttpClient y, después, agregue `ProcessURLAsync` al archivo

MainWindow.xaml.cs.

- Si desarrolló el código completando el tutorial, agregue `ProcessURLAsync` a la aplicación que usa el método `HttpClient.GetByteArrayAsync`. El archivo MainWindow.xaml.cs para esta aplicación es el segundo ejemplo en la sección "Ejemplos de código completados desde el tutorial".

El método `ProcessURLAsync` consolida las acciones en el cuerpo del bucle `foreach` de `SumPageSizesAsync` en el tutorial original. El método descarga de forma asincrónica el contenido de un sitio web especificado como una matriz de bytes y, después, muestra y devuelve la longitud de la matriz de bytes.

La única diferencia respecto al método `ProcessURLAsync` del procedimiento anterior es el uso de la instancia HttpClient, `client`.

```
async Task<int> ProcessURLAsync(string url, HttpClient client)
{
    byte[] byteArray = await client.GetByteArrayAsync(url);
    DisplayResults(url, byteArray);
    return byteArray.Length;
}
```

2. Convierta en comentario o elimine el bucle `For Each` o `foreach` de `SumPageSizesAsync`, como se muestra en el código siguiente.

```
//var total = 0;
//foreach (var url in urlList)
//{
//    // GetByteArrayAsync returns a Task<T>. At completion, the task
//    // produces a byte array.
//    byte[] urlContent = await client.GetByteArrayAsync(url);

//    // The previous line abbreviates the following two assignment
//    // statements.
//    Task<byte[]> getContentTask = client.GetByteArrayAsync(url);
//    byte[] urlContent = await getContentTask;

//    DisplayResults(url, urlContent);

//    // Update the total.
//    total += urlContent.Length;
//}
```

3. Define una consulta que, cuando la ejecute el método ToArray, crea una colección de tareas que descargan el contenido de cada sitio web. Las tareas se inician cuando se evalúa la consulta.

   Agregue el código siguiente en el método `SumPageSizesAsync` después de la declaración de `client` y `urlList`.

```
// Create a query.
IEnumerable<Task<int>> downloadTasksQuery =
    from url in urlList select ProcessURLAsync(url, client);

// Use ToArray to execute the query and start the download tasks.
Task<int>[] downloadTasks = downloadTasksQuery.ToArray();
```

4. Después, aplique `Task.WhenAll` a la colección de tareas, `downloadTasks`. `Task.WhenAll` devuelve una única tarea que finaliza cuando se completan todas las tareas de la colección de tareas.

   En el ejemplo siguiente, la expresión `await` espera la finalización de la única tarea que devuelve `WhenAll`. Cuando se completa, la expresión `await` se evalúa como una matriz de enteros, donde cada entero es la

longitud de un sitio web descargado. Agregue el código siguiente a `SumPageSizesAsync`, después del código que agregó en el paso anterior.

```
// Await the completion of all the running tasks.
int[] lengths = await Task.WhenAll(downloadTasks);

//// The previous line is equivalent to the following two statements.
//Task<int[]> whenAllTask = Task.WhenAll(downloadTasks);
//int[] lengths = await whenAllTask;
```

5. Por último, use el método Sum para obtener la suma de las longitudes de todos los sitios web. Agregue la línea siguiente a `SumPageSizesAsync`.

```
int total = lengths.Sum();
```

Para probar las soluciones Task.WhenAll

- Para cualquiera de las soluciones, presione la tecla F5 para ejecutar el programa y después haga clic en el botón **Start**. El resultado debe ser similar al de las soluciones asincrónicas en Walkthrough: Accessing the Web by Using async and await (C#) (Tutorial: acceso a la web con async y await [C#]). Pero tenga en cuenta que los sitios web cada vez aparecen en un orden diferente.

Ejemplo

El código siguiente muestra las extensiones para el proyecto que usa el método `GetURLContentsAsync` para descargar el contenido de la web.

```
// Add the following using directives, and add a reference for System.Net.Http.
using System.Net.Http;
using System.IO;
using System.Net;

namespace AsyncExampleWPF_WhenAll
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            resultsTextBox.Clear();

            // Two-step async call.
            Task sumTask = SumPageSizesAsync();
            await sumTask;

            // One-step async call.
            //await SumPageSizesAsync();

            resultsTextBox.Text += "\r\nControl returned to startButton_Click.\r\n";
        }

        private async Task SumPageSizesAsync()
        {
            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();

            // Create a query.
            IEnumerable<Task<int>> downloadTasksQuery =
```

```
        IEnumerable<Task<int>> downloadTasksQuery =
            from url in urlList select ProcessURLAsync(url);

        // Use ToArray to execute the query and start the download tasks.
        Task<int>[] downloadTasks = downloadTasksQuery.ToArray();

        // You can do other work here before awaiting.

        // Await the completion of all the running tasks.
        int[] lengths = await Task.WhenAll(downloadTasks);

        //// The previous line is equivalent to the following two statements.
        //Task<int[]> whenAllTask = Task.WhenAll(downloadTasks);
        //int[] lengths = await whenAllTask;

        int total = lengths.Sum();

        //var total = 0;
        //foreach (var url in urlList)
        //{
        //    byte[] urlContents = await GetURLContentsAsync(url);

        //    // The previous line abbreviates the following two assignment statements.
        //    // GetURLContentsAsync returns a Task<T>. At completion, the task
        //    // produces a byte array.
        //    //Task<byte[]> getContentsTask = GetURLContentsAsync(url);
        //    //byte[] urlContents = await getContentsTask;

        //    DisplayResults(url, urlContents);

        //    // Update the total.
        //    total += urlContents.Length;
        //}

        // Display the total count for all of the websites.
        resultsTextBox.Text +=
            $"\r\n\r\nTotal bytes returned:  {total}\r\n";
    }

    private List<string> SetUpURLList()
    {
        List<string> urls = new List<string>
        {
            "https://msdn.microsoft.com",
            "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
            "https://msdn.microsoft.com/library/hh290136.aspx",
            "https://msdn.microsoft.com/library/ee256749.aspx",
            "https://msdn.microsoft.com/library/hh290138.aspx",
            "https://msdn.microsoft.com/library/hh290140.aspx",
            "https://msdn.microsoft.com/library/dd470362.aspx",
            "https://msdn.microsoft.com/library/aa578028.aspx",
            "https://msdn.microsoft.com/library/ms404677.aspx",
            "https://msdn.microsoft.com/library/ff730837.aspx"
        };
        return urls;
    }

    // The actions from the foreach loop are moved to this async method.
    private async Task<int> ProcessURLAsync(string url)
    {
        var byteArray = await GetURLContentsAsync(url);
        DisplayResults(url, byteArray);
        return byteArray.Length;
    }

    private async Task<byte[]> GetURLContentsAsync(string url)
    {
        // The downloaded resource ends up in the variable named content.
        var content = new MemoryStream();
```

```
            // Initialize an HttpWebRequest for the current URL.
            var webReq = (HttpWebRequest)WebRequest.Create(url);

            // Send the request to the Internet resource and wait for
            // the response.
            using (WebResponse response = await webReq.GetResponseAsync())
            {
                // Get the data stream that is associated with the specified url.
                using (Stream responseStream = response.GetResponseStream())
                {
                    await responseStream.CopyToAsync(content);
                }
            }

            // Return the result as a byte array.
            return content.ToArray();

        }

        private void DisplayResults(string url, byte[] content)
        {
            // Display the length of each website. The string format
            // is designed to be used with a monospaced font, such as
            // Lucida Console or Global Monospace.
            var bytes = content.Length;
            // Strip off the "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

Ejemplo

El código siguiente muestra las extensiones para el proyecto que usa el método `HttpClient.GetByteArrayAsync` para descargar el contenido de la web.

```
// Add the following using directives, and add a reference for System.Net.Http.
using System.Net.Http;
using System.IO;
using System.Net;

namespace AsyncExampleWPF_HttpClient_WhenAll
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            resultsTextBox.Clear();

            // One-step async call.
            await SumPageSizesAsync();

            // Two-step async call.
            //Task sumTask = SumPageSizesAsync();
            //await sumTask;

            resultsTextBox.Text += "\r\nControl returned to startButton_Click.\r\n";
        }
```

```
        private async Task SumPageSizesAsync()
        {
            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();

            // Declare an HttpClient object and increase the buffer size. The
            // default buffer size is 65,536.
            HttpClient client = new HttpClient() { MaxResponseContentBufferSize = 1000000 };

            // Create a query.
            IEnumerable<Task<int>> downloadTasksQuery =
                from url in urlList select ProcessURLAsync(url, client);

            // Use ToArray to execute the query and start the download tasks.
            Task<int>[] downloadTasks = downloadTasksQuery.ToArray();

            // You can do other work here before awaiting.

            // Await the completion of all the running tasks.
            int[] lengths = await Task.WhenAll(downloadTasks);

            //// The previous line is equivalent to the following two statements.
            //Task<int[]> whenAllTask = Task.WhenAll(downloadTasks);
            //int[] lengths = await whenAllTask;

            int total = lengths.Sum();

            //var total = 0;
            //foreach (var url in urlList)
            //{
            //    // GetByteArrayAsync returns a Task<T>. At completion, the task
            //    // produces a byte array.
            //    byte[] urlContent = await client.GetByteArrayAsync(url);

            //    // The previous line abbreviates the following two assignment
            //    // statements.
            //    Task<byte[]> getContentTask = client.GetByteArrayAsync(url);
            //    byte[] urlContent = await getContentTask;

            //    DisplayResults(url, urlContent);

            //    // Update the total.
            //    total += urlContent.Length;
            //}

            // Display the total count for all of the web addresses.
            resultsTextBox.Text +=
                $"\r\n\r\nTotal bytes returned:  {total}\r\n";
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/hh290136.aspx",
                "https://msdn.microsoft.com/library/ee256749.aspx",
                "https://msdn.microsoft.com/library/hh290138.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/dd470362.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }

        // The actions from the foreach loop are moved to this async method.
        async Task<int> ProcessURLAsync(string url, HttpClient client)
```

```
        {
            byte[] byteArray = await client.GetByteArrayAsync(url);
            DisplayResults(url, byteArray);
            return byteArray.Length;
        }

        private void DisplayResults(string url, byte[] content)
        {
            // Display the length of each web site. The string format
            // is designed to be used with a monospaced font, such as
            // Lucida Console or Global Monospace.
            var bytes = content.Length;
            // Strip off the "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

Vea también

- Task.WhenAll
- Walkthrough: Accessing the Web by Using async and await (C#) (Tutorial: Acceso a web usando Async y Await [C#])

Procedimiento para realizar varias solicitudes web en paralelo con async y await (C#)

29/06/2020 • 9 minutes to read • Edit Online

En un método asincrónico, las tareas se inician en el momento de crearse. El operador await se aplica a la tarea en el momento del método en que no se puede continuar el procesamiento hasta que finalice la tarea. A menudo se espera una tarea en cuanto se crea, como se muestra en el ejemplo siguiente.

```
var result = await someWebAccessMethodAsync(url);
```

Pero puede separar la creación de la tarea de la espera si el programa tiene otro trabajo por efectuar que no dependa de la finalización de la tarea.

```
// The following line creates and starts the task.
var myTask = someWebAccessMethodAsync(url);

// While the task is running, you can do other work that doesn't depend
// on the results of the task.
// . . . . .

// The application of await suspends the rest of this method until the task is complete.
var result = await myTask;
```

Entre el inicio de una tarea y la espera puede iniciar otras tareas. Las tareas adicionales se ejecutan implícitamente en paralelo, pero no se crean subprocesos adicionales.

El programa siguiente inicia tres descargas web asincrónicas y las espera en el orden en el que se han llamado. Observe que, al ejecutar el programa, las tareas no siempre finalizan en el orden en que se han creado y esperado. Empiezan a ejecutarse en el momento de crearse y una o más tareas podrían finalizar antes de que el método llegue a las expresiones await.

> **NOTE**
>
> Para llevar a cabo este proyecto, debe tener instalados en el equipo Visual Studio 2012 o posterior y .NET Framework 4.5 o posterior.

Para ver otro ejemplo que inicia varias tareas al mismo tiempo, vea Procedimiento para ampliar el tutorial de async usando Task.WhenAll (C#).

Puede descargar el código de este ejemplo en Muestras de código para desarrollador.

Para configurar el proyecto

1. Para configurar una aplicación WPF, lleve a cabo los siguientes pasos. Puede encontrar instrucciones detalladas para realizar estos pasos en Tutorial: Acceso a web usando Async y Await (C#).
   - Cree una aplicación WPF que contenga un cuadro de texto y un botón. Asigne al botón el nombre `startButton` y, al cuadro de texto, `resultsTextBox`.
   - Agregue una referencia para System.Net.Http.
   - En el archivo MainWindow.xaml.cs, agregue una directiva `using` para `System.Net.Http`.

Para agregar el código

1. En la ventana de diseño, MainWindow.xaml, haga doble clic en el botón para crear el controlador de eventos `startButton_Click` en MainWindow.xaml.cs.

2. Copie el código siguiente y péguelo en el cuerpo de `startButton_Click` en MainWindow.xaml.cs.

```
resultsTextBox.Clear();
await CreateMultipleTasksAsync();
resultsTextBox.Text += "\r\n\r\nControl returned to startButton_Click.\r\n";
```

   El código llama a un método asincrónico, `CreateMultipleTasksAsync`, que dirige la aplicación.

3. Agregue los siguientes métodos de soporte técnico al proyecto:

   - `ProcessURLAsync` usa un método HttpClient para descargar el contenido de un sitio web como una matriz de bytes. Luego, el método de soporte, `ProcessURLAsync`, muestra y devuelve la longitud de la matriz.

   - `DisplayResults` muestra el número de bytes de la matriz de bytes de cada dirección URL. En esta pantalla se muestra cuándo se ha terminado la descarga de cada tarea.

   Copie los métodos siguientes y péguelos después del controlador de eventos `startButton_Click` en MainWindow.xaml.cs.

```
async Task<int> ProcessURLAsync(string url, HttpClient client)
{
    var byteArray = await client.GetByteArrayAsync(url);
    DisplayResults(url, byteArray);
    return byteArray.Length;
}

private void DisplayResults(string url, byte[] content)
{
    // Display the length of each website. The string format
    // is designed to be used with a monospaced font, such as
    // Lucida Console or Global Monospace.
    var bytes = content.Length;
    // Strip off the "https://".
    var displayURL = url.Replace("https://", "");
    resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
}
```

4. Por último, defina el método `CreateMultipleTasksAsync`, que lleva a cabo los siguientes pasos.

   - El método declara un objeto `HttpClient`, que necesita para tener acceso al método GetByteArrayAsync en `ProcessURLAsync`.

   - El método crea e inicia tres tareas de tipo Task<TResult>, donde `TResult` es un entero. A medida que finaliza cada tarea, `DisplayResults` muestra la dirección URL de la tarea y la longitud del contenido descargado. Dado que las tareas se ejecutan de manera asincrónica, el orden en que aparecen los resultados puede ser diferente del orden en que se han declarado.

   - El método espera la finalización de cada tarea. Cada operador `await` suspende la ejecución de `CreateMultipleTasksAsync` hasta que finalice la tarea esperada. El operador también recupera el valor devuelto de la llamada a `ProcessURLAsync` desde cada tarea finalizada.

   - Una vez concluidas las tareas y recuperados los valores enteros, el método suma las longitudes de los sitios web y muestra el resultado.

Copie el método siguiente y péguelo en la solución.

```
private async Task CreateMultipleTasksAsync()
{
    // Declare an HttpClient object, and increase the buffer size. The
    // default buffer size is 65,536.
    HttpClient client =
        new HttpClient() { MaxResponseContentBufferSize = 1000000 };

    // Create and start the tasks. As each task finishes, DisplayResults
    // displays its length.
    Task<int> download1 =
        ProcessURLAsync("https://msdn.microsoft.com", client);
    Task<int> download2 =
        ProcessURLAsync("https://msdn.microsoft.com/library/hh156528(VS.110).aspx", client);
    Task<int> download3 =
        ProcessURLAsync("https://msdn.microsoft.com/library/67w7t67f.aspx", client);

    // Await each task.
    int length1 = await download1;
    int length2 = await download2;
    int length3 = await download3;

    int total = length1 + length2 + length3;

    // Display the total count for the downloaded websites.
    resultsTextBox.Text += $"\r\n\r\nTotal bytes returned:  {total}\r\n";
}
```

5. Presione la tecla F5 para ejecutar el programa y elija el botón **Inicio** .

   Ejecute el programa varias veces para comprobar que las tres tareas no finalizan siempre en el mismo orden y que el orden en el que finalizan no es necesariamente el orden en que se han creado y esperado.

Ejemplo

El código siguiente contiene el ejemplo completo.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add the following using directive, and add a reference for System.Net.Http.
using System.Net.Http;

namespace AsyncExample_MultipleTasks
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
```

```
        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            resultsTextBox.Clear();
            await CreateMultipleTasksAsync();
            resultsTextBox.Text += "\r\n\r\nControl returned to startButton_Click.\r\n";
        }

        private async Task CreateMultipleTasksAsync()
        {
            // Declare an HttpClient object, and increase the buffer size. The
            // default buffer size is 65,536.
            HttpClient client =
                new HttpClient() { MaxResponseContentBufferSize = 1000000 };

            // Create and start the tasks. As each task finishes, DisplayResults
            // displays its length.
            Task<int> download1 =
                ProcessURLAsync("https://msdn.microsoft.com", client);
            Task<int> download2 =
                ProcessURLAsync("https://msdn.microsoft.com/library/hh156528(VS.110).aspx", client);
            Task<int> download3 =
                ProcessURLAsync("https://msdn.microsoft.com/library/67w7t67f.aspx", client);

            // Await each task.
            int length1 = await download1;
            int length2 = await download2;
            int length3 = await download3;

            int total = length1 + length2 + length3;

            // Display the total count for the downloaded websites.
            resultsTextBox.Text += $"\r\n\r\nTotal bytes returned:  {total}\r\n";
        }

        async Task<int> ProcessURLAsync(string url, HttpClient client)
        {
            var byteArray = await client.GetByteArrayAsync(url);
            DisplayResults(url, byteArray);
            return byteArray.Length;
        }

        private void DisplayResults(string url, byte[] content)
        {
            // Display the length of each website. The string format
            // is designed to be used with a monospaced font, such as
            // Lucida Console or Global Monospace.
            var bytes = content.Length;
            // Strip off the "https://".
            var displayURL = url.Replace("https://", "");
            resultsTextBox.Text += $"\n{displayURL,-58} {bytes,8}";
        }
    }
}
```

Vea también

- Tutorial: Acceso a web usando Async y Await (C#)
- Programación asincrónica con Async y Await (C#)
- Procedimiento para ampliar el tutorial de async usando Task.WhenAll (C#)

Tipos de valor devueltos asincrónicos (C#)

29/06/2020 • 14 minutes to read • Edit Online

Los métodos asincrónicos pueden tener los siguientes tipos de valor devuelto:

- Task<TResult>, para un método asincrónico que devuelve un valor.
- Task, para un método asincrónico que realiza una operación pero no devuelve ningún valor.
- `void`, para un controlador de eventos.
- A partir de C# 7.0, cualquier tipo que tenga un método `GetAwaiter` accesible. El objeto devuelto por el método `GetAwaiter` debe implementar la interfaz System.Runtime.CompilerServices.ICriticalNotifyCompletion.
- A partir C# 8.0, IAsyncEnumerable<T>, para un método asincrónico que devuelve una *secuencia asincrónica*.

Para más información sobre los métodos async, vea Programación asincrónica con async y await 8C#).

Tipo de valor devuelto Task<TResult>

El tipo de valor devuelto Task<TResult> se usa para un método asincrónico que contiene una instrucción return (C#) en la que el operando es `TResult`.

En el ejemplo siguiente, el método asincrónico `GetLeisureHours` contiene una instrucción `return` que devuelve un entero. Por tanto, la declaración del método debe tener un tipo de valor devuelto de `Task<int>`. El método asincrónico FromResult es un marcador de posición para una operación que devuelve una cadena.

```
public static async Task<string> ShowTodaysInfo()
{
    string ret = $"Today is {DateTime.Today:D}\n" +
                 "Today's hours of leisure: " +
                 $"{await GetLeisureHours()}";
    return ret;
}

static async Task<int> GetLeisureHours()
{
    // Task.FromResult is a placeholder for actual work that returns a string.
    var today = await Task.FromResult<string>(DateTime.Now.DayOfWeek.ToString());

    // The method then can process the result in some way.
    int leisureHours;
    if (today.First() == 'S')
        leisureHours = 16;
    else
        leisureHours = 5;

    return leisureHours;
}
// The example displays output like the following:
//    Today is Wednesday, May 24, 2017
//    Today's hours of leisure: 5
```

Cuando se llama a `GetLeisureHours` desde una expresión await en el método `ShowTodaysInfo`, esta recupera el valor entero (el valor de `leisureHours`) que está almacenado en la tarea que devuelve el método `GetLeisureHours`. Para más información sobre las expresiones await, vea await.

Puede comprender mejor cómo `await` recupera el resultado de `Task<T>` si separa la llamada a

`GetLeisureHours` de la aplicación de `await`, como se muestra en el código siguiente. Una llamada al método `GetLeisureHours` que no se espera inmediatamente devuelve `Task<int>`, como se podría esperar de la declaración del método. La tarea se asigna a la variable `integerTask` en el ejemplo. Dado que `integerTask` es Task<TResult>, contiene una propiedad Result de tipo `TResult`. En este caso, `TResult` representa un tipo entero. Cuando `await` se aplica a `integerTask`, la expresión await se evalúa en el contenido de la propiedad Result de `integerTask`. El valor se asigna a la variable `ret`.

> **IMPORTANT**
>
> La propiedad Result es una propiedad de bloqueo. Si se intenta acceder a ella antes de que termine su tarea, se bloquea el subproceso que está activo actualmente hasta que finaliza la tarea y el valor está disponible. En la mayoría de los casos, se debe tener acceso al valor usando `await` en lugar de tener acceso directamente a la propiedad.
> En el ejemplo anterior se ha recuperado el valor de la propiedad Result para bloquear el subproceso principal de manera que el método `ShowTodaysInfo` pueda terminar la ejecución antes de que finalice la aplicación.

```
var integerTask = GetLeisureHours();

// You can do other work that does not rely on integerTask before awaiting.
string ret = $"Today is {DateTime.Today:D}\n" +
             "Today's hours of leisure: " +
             $"{await integerTask}";
return ret;
```

Tipo de valor devuelto Task

Los métodos asincrónicos que no contienen una instrucción `return` o que contienen una instrucción `return` que no devuelve un operando tienen normalmente un tipo de valor devuelto de Task. Dichos métodos devuelven `void` si se ejecutan de manera sincrónica. Si se usa un tipo de valor devuelto Task para un método asincrónico, un método de llamada puede usar un operador `await` para suspender la finalización del llamador hasta que finalice el método asincrónico llamado.

En el ejemplo siguiente, el método asincrónico `WaitAndApologize` no contiene una instrucción `return`, de manera que el método devuelve un objeto Task. La devolución de `Task` permite que se espere a `WaitAndApologize`. El tipo Task no incluye una propiedad `Result` porque no tiene ningún valor devuelto.

```
public static async Task DisplayCurrentInfo()
{
    await WaitAndApologize();
    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologize()
{
    // Task.Delay is a placeholder for actual work.
    await Task.Delay(2000);
    // Task.Delay delays the following line by two seconds.
    Console.WriteLine("\nSorry for the delay. . . .\n");
}
// The example displays the following output:
//    Sorry for the delay. . . .
//
//    Today is Wednesday, May 24, 2017
//    The current time is 15:25:16.2935649
//    The current temperature is 76 degrees.
```

Se espera a `WaitAndApologize` mediante una instrucción await en lugar de una expresión await, similar a la instrucción de llamada para un método sincrónico que devuelve void. En este caso, la aplicación de un operador await no genera un valor.

Igual que en el ejemplo Task<TResult> anterior, puede separar la llamada a `WaitAndApologize` desde la aplicación de un operador await, como muestra el código siguiente. Pero recuerde que una `Task` no tiene una propiedad `Result` y que no se genera ningún valor cuando se aplica un operador await a una `Task`.

El código siguiente separa la llamada del método `WaitAndApologize` de la espera de la tarea que el método devuelve.

```
Task wait = WaitAndApologize();

string output = $"Today is {DateTime.Now:D}\n" +
                $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
                $"The current temperature is 76 degrees.\n";
await wait;
Console.WriteLine(output);
```

Tipo de valor devuelto Void

Usa el tipo de valor devuelto `void` en controladores de eventos asincrónicos, que necesitan un tipo de valor devuelto `void`. Para métodos que no sean controladores de eventos que no devuelven un valor, debe devolver Task en su lugar, porque no se espera a un método asincrónico que devuelva `void`. Cualquiera que realice la llamada a este método debe continuar hasta completarse sin esperar a que finalice el método asincrónico que se haya llamado. El autor de la llamada debe ser independiente de los valores o las excepciones que genere el método asincrónico.

El autor de la llamada de un método asincrónico que devuelve void no puede capturar las excepciones iniciadas desde el método y es probable que esas excepciones sin controlar provoquen un error de la aplicación. Si un método que devuelve un valor Task o Task<TResult> inicia una excepción, la excepción se almacena en la tarea devuelta. La excepción se vuelve a iniciar cuando se espera a la tarea. Por tanto, asegúrese de que cualquier método asincrónico que puede producir una excepción tiene un tipo de valor devuelto de Task o Task<TResult> y que se esperan llamadas al método.

Para obtener más información sobre cómo capturar excepciones en métodos asincrónicos, vea la sección Excepciones en métodos asincrónicos del artículo try-catch.

En el ejemplo siguiente se muestra el comportamiento de un controlador de eventos asincrónicos. En el código de ejemplo, un controlador de eventos asincrónicos debe informar al subproceso principal de que ha terminado. Después, el subproceso principal puede esperar a que un controlador de eventos asincrónicos finalice antes de salir del programa.

```
using System;
using System.Threading.Tasks;

public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}
```

```
public class AsyncVoidExample
{
    static TaskCompletionSource<bool> tcs = new TaskCompletionSource<bool>();

    public static async Task Main()
    {
        tcs = new TaskCompletionSource<bool>();
        var secondHandlerFinished = tcs.Task;

        var button = new NaiveButton();
        button.Clicked += Button_Clicked_1;
        button.Clicked += Button_Clicked_2_Async;
        button.Clicked += Button_Clicked_3;

        Console.WriteLine("About to click a button...");
        button.Click();
        Console.WriteLine("Button's Click method returned.");

        await secondHandlerFinished;
    }

    private static void Button_Clicked_1(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 1 is done.");
    }

    private static async void Button_Clicked_2_Async(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        tcs.SetResult(true);
    }

    private static void Button_Clicked_3(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}

// Expected output:
// About to click a button...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// Button's Click method returned.
//    Handler 2 is done.
```

Tipos de valor devueltos asincrónicos generalizados y ValueTask<TResult>

A partir de C# 7.0, un método asincrónico puede devolver cualquier tipo que tenga un método `GetAwaiter` accesible.

Como Task y Task<TResult> son tipos de referencia, la asignación de memoria en las rutas críticas para el rendimiento, especialmente cuando las asignaciones se producen en ajustados bucles, puede afectar negativamente al rendimiento. La compatibilidad para los tipos de valor devuelto generalizados significa que puede devolver un tipo de valor ligero en lugar de un tipo de referencia para evitar asignaciones de memoria adicionales.

.NET proporciona la estructura System.Threading.Tasks.ValueTask<TResult> como una implementación ligera de un valor de devolución de tareas generalizado. Para usar el tipo System.Threading.Tasks.ValueTask<TResult>, debe agregar el paquete NuGet de `System.Threading.Tasks.Extensions` a su proyecto. En el ejemplo siguiente se usa la estructura ValueTask<TResult> para recuperar el valor de dos tiradas de dado.

```
using System;
using System.Threading.Tasks;

class Program
{
   static Random? rnd;

   static async Task Main()
   {
      Console.WriteLine($"You rolled {await GetDiceRollAsync()}");
   }

   private static async ValueTask<int> GetDiceRollAsync()
   {
      Console.WriteLine("...Shaking the dice...");
      int roll1 = await RollAsync();
      int roll2 = await RollAsync();
      return roll1 + roll2;
   }

   private static async ValueTask<int> RollAsync()
   {
      if (rnd == null)
         rnd = new Random();

      await Task.Delay(500);
      int diceRoll = rnd.Next(1, 7);
      return diceRoll;
   }
}
// The example displays output like the following:
//       ...Shaking the dice...
//       You rolled 8
```

Secuencias asincrónicas con IAsyncEnumerable<T>

A partir de C# 8.0, un método asincrónico puede devolver una *secuencia asincrónica*, representada por IAsyncEnumerable<T>. Una secuencia asincrónica proporciona una manera de enumerar los elementos leídos de una secuencia cuando se generan elementos en fragmentos con llamadas asincrónicas repetidas. En el ejemplo siguiente se muestra un método asincrónico que genera una secuencia asincrónica:

```
private static async IAsyncEnumerable<string> ReadWordsFromStream()
{
    string data =
    @"This is a line of text.
      Here is the second line of text.
      And there is one more for good measure.
      Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string? line = await readStream.ReadLineAsync();
    while (line != null)
    {
        var words = line.Split(' ',StringSplitOptions.RemoveEmptyEntries);
        foreach (var word in words)
        {
            yield return word;
        }
        line = await readStream.ReadLineAsync();
    }
}
```

En el ejemplo anterior, las líneas de una cadena se leen de forma asincrónica. Una vez que se ha leído cada línea, el código enumera cada palabra de la cadena. Los autores de la llamada enumerarían cada palabra mediante la instrucción `await foreach`. El método espera cuando necesita leer de forma asincrónica la línea siguiente de la cadena de origen.

Vea también

- FromResult
- Tutorial: Acceso a web usando Async y Await (C#)
- Controlar el flujo en los programas asincrónicos (C#)
- async
- await

Controlar el flujo en los programas asincrónicos (C#)

19/03/2020 • 16 minutes to read • Edit Online

Puede escribir y mantener los programas asincrónicos más fácilmente usando las palabras clave `async` y `await` . Aun así, los resultados pueden sorprenderle si no sabe cómo funciona el programa. En este tema se hace un seguimiento del flujo de control a través de un programa asincrónico simple en el que se muestra cuándo se mueve el control de un método a otro y qué información se transfiere cada vez.

En general, los métodos que contienen código asincrónico se marcan con el modificador async (C#). En un método que está marcado con un modificador async, puede usar un operador await (C#) para especificar dónde se para el método para esperar a que concluya un proceso asincrónico al que se ha llamado. Para obtener más información, vea Programación asincrónica con async y await (C#).

En el ejemplo siguiente se usan métodos asincrónicos para descargar el contenido de un sitio web especificado como una cadena y mostrar la longitud de la cadena. El ejemplo contiene los dos métodos siguientes:

- `startButton_Click`, que llama a `AccessTheWebAsync` y muestra el resultado.
- `AccessTheWebAsync`, que descarga el contenido de un sitio web como una cadena y devuelve la longitud de esta. `AccessTheWebAsync` usa un método HttpClient asincrónico, GetStringAsync(String), para descargar el contenido.

Las líneas de visualización numeradas aparecen en puntos estratégicos de todo el programa para ayudarle a entender cómo se ejecuta el programa y explicar lo que ocurre en cada punto marcado. Las líneas de visualización tienen las etiquetas comprendidas entre "UNO" y "SEIS". Las etiquetas representan el orden en el que el programa alcanza estas líneas de código.

En el código siguiente se muestra un esquema del programa.

```
public partial class MainWindow : Window
{
    // . . .
    private async void startButton_Click(object sender, RoutedEventArgs e)
    {
        // ONE
        Task<int> getLengthTask = AccessTheWebAsync();

        // FOUR
        int contentLength = await getLengthTask;

        // SIX
        resultsTextBox.Text +=
            $"\r\nLength of the downloaded string: {contentLength}.\r\n";
    }

    async Task<int> AccessTheWebAsync()
    {
        // TWO
        HttpClient client = new HttpClient();
        Task<string> getStringTask =
            client.GetStringAsync("https://msdn.microsoft.com");

        // THREE
        string urlContents = await getStringTask;

        // FIVE
        return urlContents.Length;
    }
}
```

Cada una de las ubicaciones etiquetadas (del "UNO" al "SEIS") muestra información sobre el estado actual del programa. Se produce el siguiente resultado:

```
ONE:   Entering startButton_Click.
           Calling AccessTheWebAsync.

TWO:   Entering AccessTheWebAsync.
           Calling HttpClient.GetStringAsync.

THREE: Back in AccessTheWebAsync.
           Task getStringTask is started.
           About to await getStringTask & return a Task<int> to startButton_Click.

FOUR:  Back in startButton_Click.
           Task getLengthTask is started.
           About to await getLengthTask -- no caller to return to.

FIVE:  Back in AccessTheWebAsync.
           Task getStringTask is complete.
           Processing the return statement.
           Exiting from AccessTheWebAsync.

SIX:   Back in startButton_Click.
           Task getLengthTask is finished.
           Result from AccessTheWebAsync is stored in contentLength.
           About to display contentLength and exit.

Length of the downloaded string: 33946.
```

Configurar el programa

Puede descargar el código que se usa en este tema desde MSDN o crearlo usted mismo.

> **NOTE**
> Para ejecutar el ejemplo, debe tener instalado en el equipo Visual Studio 2012 o posterior y .NET Framework 4.5 o posterior.

Descargar el programa

Puede descargar la aplicación para este tema en Ejemplo de Async: Flujo de control en programas asincrónicos. Con los siguientes pasos se abre y se ejecuta el programa.

1. Descomprima el archivo descargado e inicie Visual Studio.
2. En la barra de menús, elija **Archivo** > **Abrir** > **Proyecto/Solución**.
3. Navegue hasta la carpeta que contiene el código de ejemplo descomprimido, abra el archivo de la solución (.sln) y presione la tecla **F5** para compilar y ejecutar el proyecto.

Crear el programa usted mismo

El siguiente proyecto de Windows Presentation Foundation (WPF) contiene el ejemplo de código de este tema.

Para ejecutar el proyecto, realice los pasos siguientes:

1. Inicie Visual Studio.
2. En la barra de menús, elija **Archivo** > **Nuevo** > **Proyecto**.

   Aparece el cuadro de diálogo **Nuevo proyecto** .
3. Elija la categoría **Instalado** > **Visual C#** > **Escritorio de Windows** y luego elija **Aplicación de WPF** en la lista de plantillas de proyecto.
4. Escriba `AsyncTracer` como el nombre del proyecto y elija el botón **Aceptar**.

   El proyecto nuevo aparece en el **Explorador de soluciones**.
5. En el Editor de código de Visual Studio, elija la pestaña **MainWindow.xaml** .

   Si la pestaña no está visible, abra el menú contextual de MainWindow.xaml en el **Explorador de soluciones** y elija **Ver código**.
6. En la vista **XAML** de MainWindow.xaml, reemplace el código por el código siguiente.

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006" mc:Ignorable="d"
x:Class="AsyncTracer.MainWindow"
        Title="Control Flow Trace" Height="350" Width="592">
    <Grid>
        <Button x:Name="startButton" Content="Start
" HorizontalAlignment="Left" Margin="250,10,0,0" VerticalAlignment="Top" Width="75" Height="24"
Click="startButton_Click" d:LayoutOverrides="GridBox"/>
        <TextBox x:Name="resultsTextBox" HorizontalAlignment="Left" TextWrapping="Wrap"
VerticalAlignment="Bottom" Width="576" Height="265" FontFamily="Lucida Console" FontSize="10"
VerticalScrollBarVisibility="Visible" Grid.ColumnSpan="3"/>
    </Grid>
</Window>
```

   En la vista **Diseño** de MainWindow.xaml aparece una ventana simple que contiene un cuadro de texto y un botón.

7. Agregue una referencia para System.Net.Http.

8. En el **Explorador de soluciones**, abra el menú contextual de MainWindow.xaml.cs y después elija **Ver código**.

9. Reemplace el código del archivo MainWindow.xaml.cs por el código siguiente.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add a using directive and a reference for System.Net.Http;
using System.Net.Http;

namespace AsyncTracer
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // The display lines in the example lead you through the control shifts.
            resultsTextBox.Text += "ONE:   Entering startButton_Click.\r\n" +
                "           Calling AccessTheWebAsync.\r\n";

            Task<int> getLengthTask = AccessTheWebAsync();

            resultsTextBox.Text += "\r\nFOUR:  Back in startButton_Click.\r\n" +
                "           Task getLengthTask is started.\r\n" +
                "           About to await getLengthTask -- no caller to return to.\r\n";

            int contentLength = await getLengthTask;

            resultsTextBox.Text += "\r\nSIX:   Back in startButton_Click.\r\n" +
                "           Task getLengthTask is finished.\r\n" +
                "           Result from AccessTheWebAsync is stored in contentLength.\r\n" +
                "           About to display contentLength and exit.\r\n";

            resultsTextBox.Text +=
                $"\r\nLength of the downloaded string: {contentLength}.\r\n";
        }

        async Task<int> AccessTheWebAsync()
        {
            resultsTextBox.Text += "\r\nTWO:   Entering AccessTheWebAsync.";

            // Declare an HttpClient object.
            HttpClient client = new HttpClient();

            resultsTextBox.Text += "\r\n           Calling HttpClient.GetStringAsync.\r\n";

            // GetStringAsync returns a Task<string>.
```

```
            Task<string> getStringTask = client.GetStringAsync("https://msdn.microsoft.com");

            resultsTextBox.Text += "\r\nTHREE: Back in AccessTheWebAsync.\r\n" +
                "           Task getStringTask is started.";

            // AccessTheWebAsync can continue to work until getStringTask is awaited.

            resultsTextBox.Text +=
                "\r\n           About to await getStringTask and return a Task<int> to
startButton_Click.\r\n";

            // Retrieve the website contents when task is complete.
            string urlContents = await getStringTask;

            resultsTextBox.Text += "\r\nFIVE:  Back in AccessTheWebAsync." +
                "\r\n           Task getStringTask is complete." +
                "\r\n           Processing the return statement." +
                "\r\n           Exiting from AccessTheWebAsync.\r\n";

            return urlContents.Length;
        }
    }
}
```

10. Presione la tecla **F5** para ejecutar el programa y elija el botón **Inicio**.

    Aparece el siguiente resultado:

```
ONE:   Entering startButton_Click.
           Calling AccessTheWebAsync.

TWO:   Entering AccessTheWebAsync.
           Calling HttpClient.GetStringAsync.

THREE: Back in AccessTheWebAsync.
           Task getStringTask is started.
           About to await getStringTask & return a Task<int> to startButton_Click.

FOUR:  Back in startButton_Click.
           Task getLengthTask is started.
           About to await getLengthTask -- no caller to return to.

FIVE:  Back in AccessTheWebAsync.
           Task getStringTask is complete.
           Processing the return statement.
           Exiting from AccessTheWebAsync.

SIX:   Back in startButton_Click.
           Task getLengthTask is finished.
           Result from AccessTheWebAsync is stored in contentLength.
           About to display contentLength and exit.

Length of the downloaded string: 33946.
```

Hacer un seguimiento del programa

Pasos UNO y DOS

Las dos primeras líneas siguen la ruta de acceso a medida que `startButton_Click` llama a `AccessTheWebAsync` y `AccessTheWebAsync` llama al método HttpClient asincrónico GetStringAsync(String). En la siguiente imagen se describen las llamadas de método a método.

```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    // The display lines in the example lead you through the control shifts.
    resultsTextBox.Text += "ONE:   Entering startButton_Click.\r\n" +
        "            Calling AccessTheWebAsync.\r\n";

    Task<int> getLengthTask = AccessTheWebAsync();
    // . . .
}


async Task<int> AccessTheWebAsync()
{
    resultsTextBox.Text += "\r\nTWO:   Entering AccessTheWebAsync.";
    // . . .
    resultsTextBox.Text += "\r\n        Calling HttpClient.GetStringAsync.\r\n";

    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");
    // . . .
}
```

```
Task<string> HttpClient.GetStringAsync (string url)
```

El tipo de valor devuelto de `AccessTheWebAsync` y `client.GetStringAsync` es Task<TResult>. Para `AccessTheWebAsync`, TResult es un entero. Para `GetStringAsync`, TResult es una cadena. Para obtener más información sobre los tipos de valor devuelto de los métodos asincrónicos, vea Tipos de valor devueltos asincrónicos (C#).

Un método asincrónico de devolución de tarea devuelve una instancia de la tarea cuando el control se desplaza al llamador. El control vuelve a su llamador procedente de un método asincrónico cuando se encuentra un operador `await` en el método llamado o cuando este finaliza. Las líneas de visualización que tienen las etiquetas comprendidas entre "TRES" y "SEIS" rastrean esta parte del proceso.

Paso TRES

En `AccessTheWebAsync`, el método asincrónico GetStringAsync(String) se llama para descargar el contenido de la página web de destino. El control vuelve a `AccessTheWebAsync` procedente de `client.GetStringAsync` cuando se devuelve `client.GetStringAsync`.

El método `client.GetStringAsync` devuelve una tarea de la cadena que se asigna a la variable `getStringTask` en `AccessTheWebAsync`. En la siguiente línea del programa de ejemplo se muestra la llamada a `client.GetStringAsync` y la asignación.

```
Task<string> getStringTask = client.GetStringAsync("https://msdn.microsoft.com");
```

Puede considerar la tarea como una promesa de `client.GetStringAsync` de generar una cadena real. Mientras tanto, si `AccessTheWebAsync` tiene trabajo que no depende de la cadena prometida de `client.GetStringAsync`, dicho trabajo puede continuar mientras `client.GetStringAsync` espera. En el ejemplo, las siguientes líneas de salida, que tienen la etiqueta "TRES", representan la oportunidad de realizar un trabajo independiente

```
THREE: Back in AccessTheWebAsync.
           Task getStringTask is started.
           About to await getStringTask & return a Task<int> to startButton_Click.
```

La siguiente instrucción suspende el progreso en `AccessTheWebAsync` cuando se espera a `getStringTask`.

```
string urlContents = await getStringTask;
```

En la siguiente imagen se muestra el flujo de control procedente de `client.GetStringAsync` a la asignación de `getStringTask` y procedente de la creación de `getStringTask` a la aplicación de un operador await.

```
async Task<int> AccessTheWebAsync()
{
    // . . .

    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    resultsTextBox.Text += "\r\nTHREE: Back in AccessTheWebAsync.\r\n" +
        "           Task getStringTask is started.";

    // AccessTheWebAsync can continue to work until getStringTask is awaited.

    string urlContents = await getStringTask;

    // . . .

    return urlContents.Length;
}
```

```
Task<string> HttpClient.GetStringAsync (string url)
```

La expresión await suspende `AccessTheWebAsync` hasta que se devuelva `client.GetStringAsync`. Mientras tanto, el control vuelve al llamador de `AccessTheWebAsync`, `startButton_Click`.

> **NOTE**
>
> Normalmente se espera la llamada a un método asincrónico de forma inmediata. Por ejemplo, la siguiente asignación podría reemplazar el código anterior que crea y espera `getStringTask`:
>
> `string urlContents = await client.GetStringAsync("https://msdn.microsoft.com");`
>
> En este tema, el operador await se aplica más adelante para dar cabida a las líneas de salida que marcan el flujo de control a través del programa.

Paso CUATRO

El tipo de valor devuelto declarado de `AccessTheWebAsync` es `Task<int>`. Por lo tanto, cuando se suspende `AccessTheWebAsync`, devuelve una tarea de entero en `startButton_Click`. Debe entender que la tarea devuelta no es `getStringTask`. La tarea devuelta es una nueva tarea de entero que representa lo que falta por hacer en el método suspendido, `AccessTheWebAsync`. La tarea es una promesa de `AccessTheWebAsync` de generar un entero cuando finalice la tarea.

La siguiente instrucción asigna esta tarea a la variable `getLengthTask`.

```
Task<int> getLengthTask = AccessTheWebAsync();
```

Como en `AccessTheWebAsync`, `startButton_Click` puede continuar con el trabajo que no depende de los resultados de la tarea asincrónica (`getLengthTask`) hasta que se espere la tarea. Las siguientes líneas de salida representan ese trabajo.

```
FOUR:  Back in startButton_Click.
           Task getLengthTask is started.
           About to await getLengthTask -- no caller to return to.
```

El progreso de `startButton_Click` se suspende cuando se espera `getLengthTask`. La siguiente instrucción de asignación suspende `startButton_Click` hasta que concluya `AccessTheWebAsync`.

```
int contentLength = await getLengthTask;
```

En la siguiente ilustración, las flechas muestran el flujo de control desde la expresión await en `AccessTheWebAsync` hasta la asignación de un valor a `getLengthTask`, seguido del procesamiento normal en `startButton_Click` hasta que se espera a `getLengthTask`.

```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    // . . .

    Task<int> getLengthTask = AccessTheWebAsync();

    resultsTextBox.Text += "\r\nFOUR:  Back in startButton_Click.\r\n" +
        "            Task getLengthTask is started.";

    int contentLength = await getLengthTask;

    // . . .

    resultsTextBox.Text +=
        String.Format("\r\nLength of the downloaded string: {0}.\r\n", contentLength);
}


async Task<int> AccessTheWebAsync()
{
    // . . .

    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    resultsTextBox.Text += "\r\nTHREE: Back in AccessTheWebAsync.\r\n" +
        "            Task getStringTask is started.";

    // AccessTheWebAsync can continue to work until getStringTask is awaited.

    string urlContents = await getStringTask;

    // . . .

    return urlContents.Length;
}
```

Paso CINCO

Cuando `client.GetStringAsync` indica que ha finalizado, el procesamiento de `AccessTheWebAsync` sale de la suspensión y puede continuar una vez superada la instrucción await. En las siguientes líneas de salida se representa la reanudación del procesamiento.

```
FIVE:  Back in AccessTheWebAsync.
           Task getStringTask is complete.
           Processing the return statement.
           Exiting from AccessTheWebAsync.
```

El operando de la instrucción de devolución, `urlContents.Length`, se almacena en la tarea que devuelve `AccessTheWebAsync`. La expresión await recupera ese valor de `getLengthTask` en `startButton_Click`.

En la siguiente imagen se muestra la transferencia de control una vez concluido `client.GetStringAsync` (y `getStringTask`).

```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    Task<int> getLengthTask = AccessTheWebAsync();

    int contentLength = await getLengthTask;

    resultsTextBox.Text += "\r\nSIX:   Back in startButton_Click. . . .\r\n";

    resultsTextBox.Text +=
        String.Format("\r\nLength of the downloaded string: {0}.\r\n", contentLength);
}


async Task<int> AccessTheWebAsync()
{
    // . . .
    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    resultsTextBox.Text += "\r\nTHREE: Back in AccessTheWebAsync.\r\n" +
        "            Task getStringTask is started.";

    // Retrieve the website contents when task is complete.
    string urlContents = await getStringTask;

    resultsTextBox.Text += "\r\nFIVE:  Back in AccessTheWebAsync." +
        "\r\n            Task getStringTask is complete." +
        "\r\n            Processing the return statement." +
        "\r\n            Exiting from AccessTheWebAsync.\r\n";

    return urlContents.Length;
}
```

`AccessTheWebAsync` se ejecuta hasta el final y el control vuelve a `startButton_Click`, que espera la finalización.

Paso SEIS

Cuando `AccessTheWebAsync` indica que ha finalizado, el procesamiento puede continuar una vez superada la instrucción await en `startButton_Async`. De hecho, el programa no tiene nada más que hacer.

En las siguientes líneas de salida se representa la reanudación del procesamiento en `startButton_Async`:

```
SIX:   Back in startButton_Click.
           Task getLengthTask is finished.
           Result from AccessTheWebAsync is stored in contentLength.
           About to display contentLength and exit.
```

La expresión await recupera de `getLengthTask` el valor entero que es el operando de la instrucción de devolución de `AccessTheWebAsync`. La siguiente instrucción asigna ese valor a la variable `contentLength`.

```
int contentLength = await getLengthTask;
```

En la siguiente imagen se muestra la devolución del control de `AccessTheWebAsync` a `startButton_Click`.

```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    Task<int> getLengthTask = AccessTheWebAsync();

    // . . .

    int contentLength = await getLengthTask;

    resultsTextBox.Text += "\r\nSIX:   Back in startButton_Click.\r\n" +
        "            Task getLengthTask is finished.\r\n" +
        "            Result from AccessTheWebAsync is stored in contentLength.\r\n" +
        "            About to display contentLength and exit.\r\n";

    resultsTextBox.Text +=
        String.Format("\r\nLength of the downloaded string: {0}.\r\n", contentLength);
}


async Task<int> AccessTheWebAsync()
{
    // . . .
    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");
    // . . .

    // Retrieve the website contents when task is complete.
    string urlContents = await getStringTask;

    resultsTextBox.Text += "\r\nFIVE:  Back in AccessTheWebAsync. . . .r\n";

    return urlContents.Length;
}
```

Vea también

- Programación asincrónica con Async y Await (C#)
- Tipos de valor devueltos asincrónicos (C#)
- Tutorial: Acceso a web usando Async y Await (C#)
- Ejemplo de async: Flujo de control en programas asincrónicos (C# y Visual Basic)

Ajustar una aplicación asincrónica (C#)

18/03/2020 • 2 minutes to read • Edit Online

Puede agregar precisión y flexibilidad a sus aplicaciones asincrónicas mediante los métodos y las propiedades que el tipo Task pone a su disposición. Los temas de esta sección muestran ejemplos que usan CancellationToken y métodos `Task` importantes como Task.WhenAll y Task.WhenAny.

Con `WhenAny` y `WhenAll`, puede iniciar más fácilmente varias tareas y esperar su finalización mediante la supervisión de una sola tarea.

- `WhenAny` devuelve una tarea que se completa cuando se complete cualquier tarea de una colección.

  Para obtener ejemplos del uso de `WhenAny`, vea Cancelar las tareas asincrónicas restantes cuando se completa una (C#) y Iniciar varias tareas asincrónicas y procesarlas a medida que se completan (C#).

- `WhenAll` devuelve una tarea que se completa cuando se completen todas las tareas de una colección.

  Para obtener más información y un ejemplo del uso de `WhenAll`, vea Procedimiento para ampliar el tutorial de async usando Task.WhenAll (C#).

Esta sección contiene los siguientes ejemplos:

- Cancelar una tarea asincrónica o una lista de tareas (C#).
- Cancelar tareas asincrónicas tras un período de tiempo (C#)
- Cancelar las tareas asincrónicas restantes cuando se completa una (C#)
- Iniciar varias tareas asincrónicas y procesarlas a medida que se completan (C#)

> **NOTE**
> Para ejecutar los ejemplos, debe tener Visual Studio 2012 o posterior, y .NET Framework 4.5 o posterior, instalado en el equipo.

Los proyectos crean una interfaz de usuario que contiene un botón que inicia el proceso y un botón que lo cancela, tal como se muestra en la imagen siguiente. Los botones se denominan `startButton` y `cancelButton`.

Puede descargar los proyectos completos de Windows Presentation Foundation (WPF) de Async Sample: Fine Tuning Your Application (Ejemplo Async: Ajustar la aplicación).

Vea también

- Asynchronous Programming with async and await (C#) (Programación asincrónica con async y await (C#))

Cancelar una tarea asincrónica o una lista de tareas (C#)

18/03/2020 • 15 minutes to read • Edit Online

Puede configurar un botón para cancelar una aplicación asincrónica si no quiere esperar a que termine. Mediante los ejemplos de este tema, puede agregar un botón de cancelación a una aplicación que descargue el contenido de un sitio web o una lista de sitios web.

En los ejemplos se usa la interfaz de usuario que se describe en Ajustar una aplicación asincrónica (C#).

> **NOTE**
> Para ejecutar los ejemplos, debe tener Visual Studio 2012 o posterior, y .NET Framework 4.5 o posterior, instalado en el equipo.

Cancelar una tarea

En el primer ejemplo se asocia el botón **Cancelar** a una sola tarea de descarga. Si elige el botón mientras la aplicación descarga contenido, se cancela la descarga.

Descargar el ejemplo

Puede descargar el proyecto completo de Windows Presentation Foundation (WPF) en Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]) y después seguir estos pasos.

1. Descomprima el archivo descargado y, a continuación, inicie Visual Studio.
2. En la barra de menús, elija **Archivo** > **Abrir** > **Proyecto/Solución**.
3. En el cuadro de diálogo **Abrir proyecto**, abra la carpeta que contiene el código de ejemplo que ha descomprimido y, después, abra el archivo de solución (.sln) para AsyncFineTuningCS.
4. En el **Explorador de soluciones**, abra el menú contextual del proyecto **CancelATask** y, después, elija **Establecer como proyecto de inicio**.
5. Presione la tecla **F5** para ejecutar el proyecto, o bien presione las teclas **Ctrl+F5** para ejecutar el proyecto sin depurarlo.

> **TIP**
> Si no quiere descargar el proyecto, puede revisar los archivos MainWindow.xaml.cs al final de este tema.

Compilar el ejemplo

Los cambios siguientes agregan un botón **Cancelar** a una aplicación que descarga un sitio web. Si no quiere descargar ni generar el ejemplo, puede revisar el producto final en la sección "Ejemplos completos" al final de este tema. Los cambios en el código se marcan con asteriscos.

Para generar su propio ejemplo, paso a paso, siga las instrucciones de la sección "Descargar el ejemplo", pero elija **StarterCode** como **Proyecto de inicio** en lugar de **CancelATask**.

Luego, realice los cambios siguientes en el archivo MainWindow.xaml.cs del proyecto.

1. Declare una variable de `CancellationTokenSource`, `cts`, que esté en el ámbito de todos los métodos que acceden a ella.

```
public partial class MainWindow : Window
{
    // ***Declare a System.Threading.CancellationTokenSource.
    CancellationTokenSource cts;
```

2. Agregue el controlador de eventos siguiente para el botón **Cancelar**. El controlador de eventos usa el método CancellationTokenSource.Cancel para notificar `cts` cuando el usuario solicita la cancelación.

```
// ***Add an event handler for the Cancel button.
private void cancelButton_Click(object sender, RoutedEventArgs e)
{
    if (cts != null)
    {
        cts.Cancel();
    }
}
```

3. Realice los siguientes cambios en el controlador de eventos para el botón **Iniciar**, `startButton_Click`.

   - Cree una instancia de `CancellationTokenSource`, `cts`.

```
// ***Instantiate the CancellationTokenSource.
cts = new CancellationTokenSource();
```

   - En la llamada a `AccessTheWebAsync`, que descarga el contenido de un sitio web especificado, envíe la propiedad CancellationTokenSource.Token de `cts` como un argumento. La propiedad `Token` propaga el mensaje si se solicita la cancelación. Agregue un bloque catch que muestre un mensaje si el usuario decide cancelar la operación de descarga. En el código siguiente se muestran los cambios.

```
try
{
    // ***Send a token to carry the message if cancellation is requested.
    int contentLength = await AccessTheWebAsync(cts.Token);
    resultsTextBox.Text += $"\r\nLength of the downloaded string: {contentLength}.\r\n";
}
// *** If cancellation is requested, an OperationCanceledException results.
catch (OperationCanceledException)
{
    resultsTextBox.Text += "\r\nDownload canceled.\r\n";
}
catch (Exception)
{
    resultsTextBox.Text += "\r\nDownload failed.\r\n";
}
```

4. En `AccessTheWebAsync`, use la sobrecarga HttpClient.GetAsync(String, CancellationToken) del método `GetAsync` en el tipo HttpClient para descargar el contenido de un sitio web. Pase `ct`, el parámetro CancellationToken de `AccessTheWebAsync`, como el segundo argumento. El token lleva el mensaje si el usuario elige el botón **Cancelar**.

   En el código siguiente se muestran los cambios en `AccessTheWebAsync`.

```
// ***Provide a parameter for the CancellationToken.
async Task<int> AccessTheWebAsync(CancellationToken ct)
{
    HttpClient client = new HttpClient();

    resultsTextBox.Text += "\r\nReady to download.\r\n";

    // You might need to slow things down to have a chance to cancel.
    await Task.Delay(250);

    // GetAsync returns a Task<HttpResponseMessage>.
    // ***The ct argument carries the message if the Cancel button is chosen.
    HttpResponseMessage response = await
client.GetAsync("https://msdn.microsoft.com/library/dd470362.aspx", ct);

    // Retrieve the website contents from the HttpResponseMessage.
    byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

    // The result of the method is the length of the downloaded website.
    return urlContents.Length;
}
```

5. Si no lo cancela, el programa produce el resultado siguiente.

```
Ready to download.
Length of the downloaded string: 158125.
```

Si elige el botón **Cancelar** antes de que el programa termine de descargar el contenido, el programa produce el resultado siguiente.

```
Ready to download.
Download canceled.
```

Cancelar una lista de tareas

Puede ampliar el ejemplo anterior para cancelar muchas tareas asociando la misma instancia de `CancellationTokenSource` a cada tarea. Si elige el botón **Cancelar**, cancela todas las tareas que aún no se han completado.

Descargar el ejemplo

Puede descargar el proyecto completo de Windows Presentation Foundation (WPF) en Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]) y después seguir estos pasos.

1. Descomprima el archivo descargado y, a continuación, inicie Visual Studio.

2. En la barra de menús, elija **Archivo** > **Abrir** > **Proyecto/Solución**.

3. En el cuadro de diálogo **Abrir proyecto**, abra la carpeta que contiene el código de ejemplo que ha descomprimido y, después, abra el archivo de solución (.sln) para AsyncFineTuningCS.

4. En el **Explorador de soluciones**, abra el menú contextual del proyecto **CancelAListOfTasks** y, después, elija **Establecer como proyecto de inicio**.

5. Presione la tecla **F5** para ejecutar el proyecto.

   Presione las teclas **Ctrl+F5** para ejecutar el proyecto sin depurarlo.

Si no quiere descargar el proyecto, puede revisar los archivos MainWindow.xaml.cs al final de este tema.

Compilar el ejemplo

Para ampliar el ejemplo personalmente, paso a paso, siga las instrucciones de la sección "Descargar el ejemplo", pero elija **CancelATask** como el **Proyecto de inicio**. Agregue los siguientes cambios a ese proyecto. Los cambios en el programa se marcan con asteriscos.

1. Agregue un método para crear una lista de direcciones web.

```
// ***Add a method that creates a list of web addresses.
private List<string> SetUpURLList()
{
    List<string> urls = new List<string>
    {
        "https://msdn.microsoft.com",
        "https://msdn.microsoft.com/library/hh290138.aspx",
        "https://msdn.microsoft.com/library/hh290140.aspx",
        "https://msdn.microsoft.com/library/dd470362.aspx",
        "https://msdn.microsoft.com/library/aa578028.aspx",
        "https://msdn.microsoft.com/library/ms404677.aspx",
        "https://msdn.microsoft.com/library/ff730837.aspx"
    };
    return urls;
}
```

2. Llame al método en `AccessTheWebAsync`.

```
// ***Call SetUpURLList to make a list of web addresses.
List<string> urlList = SetUpURLList();
```

3. Agregue el siguiente bucle en `AccessTheWebAsync` para procesar cada dirección web de la lista.

```
// ***Add a loop to process the list of web addresses.
foreach (var url in urlList)
{
    // GetAsync returns a Task<HttpResponseMessage>.
    // Argument ct carries the message if the Cancel button is chosen.
    // ***Note that the Cancel button can cancel all remaining downloads.
    HttpResponseMessage response = await client.GetAsync(url, ct);

    // Retrieve the website contents from the HttpResponseMessage.
    byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

    resultsTextBox.Text +=
        $"\r\nLength of the downloaded string: {urlContents.Length}.\r\n";
}
```

4. Como `AccessTheWebAsync` muestra las duraciones, el método no tiene que devolver nada. Quite la instrucción Return y cambie el tipo de valor devuelto del método a Task en lugar de Task<TResult>.

```
async Task AccessTheWebAsync(CancellationToken ct)
```

   Llame al método desde `startButton_Click` mediante una instrucción en lugar de una expresión.

```
await AccessTheWebAsync(cts.Token);
```

5. Si no lo cancela, el programa produce el resultado siguiente.

```
Length of the downloaded string: 35939.

Length of the downloaded string: 237682.

Length of the downloaded string: 128607.

Length of the downloaded string: 158124.

Length of the downloaded string: 204890.

Length of the downloaded string: 175488.

Length of the downloaded string: 145790.

Downloads complete.
```

Si elige el botón **Cancelar** antes de que se completen las descargas, la salida contiene las duraciones de las descargas completadas antes de la cancelación.

```
Length of the downloaded string: 35939.

Length of the downloaded string: 237682.

Length of the downloaded string: 128607.

Downloads canceled.
```

Ejemplos completos

Las secciones siguientes contienen el código para cada uno de los ejemplos anteriores. Observe que debe agregar una referencia para System.Net.Http.

Puede descargar los proyectos en Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]).

Ejemplo: Cancelar una tarea

El código siguiente es el archivo MainWindow.xaml.cs completo del ejemplo que cancela una sola tarea.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add a using directive and a reference for System.Net.Http.
using System.Net.Http;

// Add the following using directive for System.Threading.

using System.Threading;
namespace CancelATask
{
    public partial class MainWindow : Window
```

```
    public partial class MainWindow : Window
    {
        // ***Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // ***Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            resultsTextBox.Clear();

            try
            {
                // ***Send a token to carry the message if cancellation is requested.
                int contentLength = await AccessTheWebAsync(cts.Token);
                resultsTextBox.Text +=
                    $"\r\nLength of the downloaded string: {contentLength}.\r\n";
            }
            // *** If cancellation is requested, an OperationCanceledException results.
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownload canceled.\r\n";
            }
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownload failed.\r\n";
            }

            // ***Set the CancellationTokenSource to null when the download is complete.
            cts = null;
        }

        // ***Add an event handler for the Cancel button.
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        // ***Provide a parameter for the CancellationToken.
        async Task<int> AccessTheWebAsync(CancellationToken ct)
        {
            HttpClient client = new HttpClient();

            resultsTextBox.Text += "\r\nReady to download.\r\n";

            // You might need to slow things down to have a chance to cancel.
            await Task.Delay(250);

            // GetAsync returns a Task<HttpResponseMessage>.
            // ***The ct argument carries the message if the Cancel button is chosen.
            HttpResponseMessage response = await
client.GetAsync("https://msdn.microsoft.com/library/dd470362.aspx", ct);

            // Retrieve the website contents from the HttpResponseMessage.
            byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

            // The result of the method is the length of the downloaded website.
            return urlContents.Length;
        }
    }
```

```
    // Output for a successful download:

    // Ready to download.

    // Length of the downloaded string: 158125.

    // Or, if you cancel:

    // Ready to download.

    // Download canceled.
}
```

Ejemplo: Cancelar una lista de tareas

El código siguiente es el archivo MainWindow.xaml.cs completo del ejemplo que cancela una lista de tareas.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add a using directive and a reference for System.Net.Http.
using System.Net.Http;

// Add the following using directive for System.Threading.
using System.Threading;

namespace CancelAListOfTasks
{
    public partial class MainWindow : Window
    {
        // Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            resultsTextBox.Clear();

            try
            {
                await AccessTheWebAsync(cts.Token);
                // ***Small change in the display lines.
                resultsTextBox.Text += "\r\nDownloads complete.";
            }
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownloads canceled.";
            }
```

```
        }
        catch (Exception)
        {
            resultsTextBox.Text += "\r\nDownloads failed.";
        }

        // Set the CancellationTokenSource to null when the download is complete.
        cts = null;
    }

    // Add an event handler for the Cancel button.
    private void cancelButton_Click(object sender, RoutedEventArgs e)
    {
        if (cts != null)
        {
            cts.Cancel();
        }
    }

    // Provide a parameter for the CancellationToken.
    // ***Change the return type to Task because the method has no return statement.
    async Task AccessTheWebAsync(CancellationToken ct)
    {
        // Declare an HttpClient object.
        HttpClient client = new HttpClient();

        // ***Call SetUpURLList to make a list of web addresses.
        List<string> urlList = SetUpURLList();

        // ***Add a loop to process the list of web addresses.
        foreach (var url in urlList)
        {
            // GetAsync returns a Task<HttpResponseMessage>.
            // Argument ct carries the message if the Cancel button is chosen.
            // ***Note that the Cancel button can cancel all remaining downloads.
            HttpResponseMessage response = await client.GetAsync(url, ct);

            // Retrieve the website contents from the HttpResponseMessage.
            byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

            resultsTextBox.Text +=
                $"\r\nLength of the downloaded string: {urlContents.Length}.\r\n";
        }
    }

    // ***Add a method that creates a list of web addresses.
    private List<string> SetUpURLList()
    {
        List<string> urls = new List<string>
        {
            "https://msdn.microsoft.com",
            "https://msdn.microsoft.com/library/hh290138.aspx",
            "https://msdn.microsoft.com/library/hh290140.aspx",
            "https://msdn.microsoft.com/library/dd470362.aspx",
            "https://msdn.microsoft.com/library/aa578028.aspx",
            "https://msdn.microsoft.com/library/ms404677.aspx",
            "https://msdn.microsoft.com/library/ff730837.aspx"
        };
        return urls;
    }
}

// Output if you do not choose to cancel:

//Length of the downloaded string: 35939.

//Length of the downloaded string: 237682.

//Length of the downloaded string: 128607.
```

```
    //Length of the downloaded string: 158124.

    //Length of the downloaded string: 204890.

    //Length of the downloaded string: 175488.

    //Length of the downloaded string: 145790.

    //Downloads complete.

    // Sample output if you choose to cancel:

    //Length of the downloaded string: 35939.

    //Length of the downloaded string: 237682.

    //Length of the downloaded string: 128607.

    //Downloads canceled.
}
```

Vea también

- CancellationTokenSource
- CancellationToken
- Asynchronous Programming with async and await (C#) (Programación asincrónica con async y await (C#))
- Fine-Tuning Your Async Application (C#) (Ajuste de la aplicación asincrónica [C#])
- Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación)

Cancelar tareas asincrónicas tras un período de tiempo (C#)

18/03/2020 • 7 minutes to read • Edit Online

Puede cancelar una operación asincrónica después de un período de tiempo con el método CancellationTokenSource.CancelAfter si no quiere esperar a que finalice la operación. Este método programa la cancelación de las tareas asociadas que no se completen en el período de tiempo designado por la expresión `CancelAfter`.

Este ejemplo se agrega al código que se desarrolla en Cancelar una tarea asincrónica o una lista de tareas (C#) para descargar una lista de sitios web y mostrar la longitud del contenido de cada uno de ellos.

> **NOTE**
>
> Para ejecutar los ejemplos, debe tener Visual Studio 2012 o posterior, y .NET Framework 4.5 o posterior, instalado en el equipo.

Descargar el ejemplo

Puede descargar el proyecto completo de Windows Presentation Foundation (WPF) en Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]) y después seguir estos pasos.

1. Descomprima el archivo descargado y, a continuación, inicie Visual Studio.
2. En la barra de menús, elija **Archivo** > **Abrir** > **Proyecto/Solución**.
3. En el cuadro de diálogo **Abrir proyecto**, abra la carpeta que contiene el código de ejemplo que ha descomprimido y, después, abra el archivo de solución (.sln) para AsyncFineTuningCS.
4. En el **Explorador de soluciones**, abra el menú contextual del proyecto **CancelAfterTime** y, después, elija **Establecer como proyecto de inicio**.
5. Presione la tecla **F5** para ejecutar el proyecto. (O presione **Ctrl+F5** para ejecutar el proyecto sin depurarlo).
6. Ejecute el programa varias veces para comprobar que la salida puede mostrar resultados para todos los sitios web, para ningún sitio web o para algunos sitios web.

Si no quiere descargar el proyecto, puede revisar el archivo MainWindow.xaml.cs al final de este tema.

Compilar el ejemplo

El ejemplo de este tema se incorpora al proyecto desarrollado en Cancel an Async Task or a List of Tasks (C#) (Cancelar una tarea asincrónica o una lista de tareas [C#]) para cancelar una lista de tareas. En el ejemplo se usa la misma interfaz de usuario, aunque el botón **Cancelar** no se usa explícitamente.

Para generar su propio ejemplo, paso a paso, siga las instrucciones de la sección "Descargar el ejemplo", pero elija **CancelAListOfTasks** como **Proyecto de inicio**. Agregue los cambios de este tema a ese proyecto.

Para especificar un tiempo máximo antes de que las tareas se marquen como canceladas, agregue una llamada a `CancelAfter` en `startButton_Click`, tal y como se muestra en el ejemplo siguiente. La adición se marca con asteriscos.

```
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    // Instantiate the CancellationTokenSource.
    cts = new CancellationTokenSource();

    resultsTextBox.Clear();

    try
    {
        // ***Set up the CancellationTokenSource to cancel after 2.5 seconds. (You
        // can adjust the time.)
        cts.CancelAfter(2500);

        await AccessTheWebAsync(cts.Token);
        resultsTextBox.Text += "\r\nDownloads succeeded.\r\n";
    }
    catch (OperationCanceledException)
    {
        resultsTextBox.Text += "\r\nDownloads canceled.\r\n";
    }
    catch (Exception)
    {
        resultsTextBox.Text += "\r\nDownloads failed.\r\n";
    }

    cts = null;
}
```

Ejecute el programa varias veces para comprobar que la salida puede mostrar resultados para todos los sitios web, para ningún sitio web o para algunos sitios web. El siguiente resultado es un ejemplo.

```
Length of the downloaded string: 35990.

Length of the downloaded string: 407399.

Length of the downloaded string: 226091.

Downloads canceled.
```

Ejemplo completo

El código siguiente es el texto completo del archivo MainWindow.xaml.cs para el ejemplo. Los asteriscos marcan los elementos que se han agregado a este ejemplo.

Observe que debe agregar una referencia para System.Net.Http.

Puede descargar el proyecto de Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
```

```
using System.Windows.Shapes;

// Add a using directive and a reference for System.Net.Http.
using System.Net.Http;

// Add the following using directive.
using System.Threading;

namespace CancelAfterTime
{
    public partial class MainWindow : Window
    {
        // Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            resultsTextBox.Clear();

            try
            {
                // ***Set up the CancellationTokenSource to cancel after 2.5 seconds. (You
                // can adjust the time.)
                cts.CancelAfter(2500);

                await AccessTheWebAsync(cts.Token);
                resultsTextBox.Text += "\r\nDownloads succeeded.\r\n";
            }
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownloads canceled.\r\n";
            }
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownloads failed.\r\n";
            }

            cts = null;
        }

        // You can still include a Cancel button if you want to.
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        async Task AccessTheWebAsync(CancellationToken ct)
        {
            // Declare an HttpClient object.
            HttpClient client = new HttpClient();

            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();

            foreach (var url in urlList)
            {
                // GetAsync returns a Task<HttpResponseMessage>.
                // Argument ct carries the message if the Cancel button is chosen.
```

```
                // Note that the Cancel button cancels all remaining downloads.
                HttpResponseMessage response = await client.GetAsync(url, ct);

                // Retrieve the website contents from the HttpResponseMessage.
                byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

                resultsTextBox.Text +=
                    $"\r\nLength of the downloaded string: {urlContents.Length}.\r\n";
            }
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                "https://msdn.microsoft.com/library/hh290136.aspx",
                "https://msdn.microsoft.com/library/ee256749.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }
    }

    // Sample Output:

    // Length of the downloaded string: 35990.

    // Length of the downloaded string: 407399.

    // Length of the downloaded string: 226091.

    // Downloads canceled.
}
```

Vea también

- Asynchronous Programming with async and await (C#) (Programación asincrónica con async y await (C#))
- Walkthrough: Accessing the Web by Using async and await (C#) (Tutorial: Acceso a web usando Async y Await [C#])
- Cancelar una tarea asincrónica o una lista de tareas (C#)
- Fine-Tuning Your Async Application (C#) (Ajuste de la aplicación asincrónica [C#])
- Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación)

Cancelar las tareas asincrónicas restantes cuando se completa una (C#)

18/03/2020 • 10 minutes to read • Edit Online

Mediante el método Task.WhenAny junto con CancellationToken, puede cancelar todas las tareas restantes cuando se completa una tarea. El método `WhenAny` toma un argumento que es una colección de tareas. El método inicia todas las tareas y devuelve una sola tarea. La tarea se completa cuando se complete cualquier tarea de la colección.

En este ejemplo se muestra cómo usar un token de cancelación junto con `WhenAny` para retener la primera tarea para finalizar de la colección de tareas y cancelar las tareas restantes. Cada tarea descarga el contenido de un sitio web. En el ejemplo se muestra la longitud del contenido de la primera descarga completa y se cancelan las otras descargas.

> **NOTE**
> Para ejecutar los ejemplos, debe tener Visual Studio 2012 o posterior, y .NET Framework 4.5 o posterior, instalado en el equipo.

Descargar el ejemplo

Puede descargar el proyecto completo de Windows Presentation Foundation (WPF) en Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]) y después seguir estos pasos.

1. Descomprima el archivo descargado y, a continuación, inicie Visual Studio.
2. En la barra de menús, elija **Archivo**, **Abrir**, **Proyecto o solución**.
3. En el cuadro de diálogo **Abrir proyecto**, abra la carpeta que contiene el código de ejemplo que ha descomprimido y, después, abra el archivo de solución (.sln) para AsyncFineTuningCS.
4. En el **Explorador de soluciones**, abra el menú contextual del proyecto **CancelAfterOneTask** y, después, elija **Establecer como proyecto de inicio**.
5. Pulse la tecla F5 para ejecutar el proyecto.

   Pulse las teclas Ctrl+F5 para ejecutar el proyecto sin depurarlo.
6. Ejecute el programa varias veces para comprobar que finalizan primero descargas diferentes.

Si no quiere descargar el proyecto, puede revisar el archivo MainWindow.xaml.cs al final de este tema.

Compilar el ejemplo

El ejemplo de este tema se incorpora al proyecto desarrollado en Cancel an Async Task or a List of Tasks (C#) (Cancelar una tarea asincrónica o una lista de tareas [C#]) para cancelar una lista de tareas. En el ejemplo se usa la misma interfaz de usuario, aunque el botón **Cancelar** no se usa explícitamente.

Para generar su propio ejemplo, paso a paso, siga las instrucciones de la sección "Descargar el ejemplo", pero elija **CancelAListOfTasks** como **Proyecto de inicio**. Agregue los cambios de este tema a ese proyecto.

En el archivo MainWindow.xaml.cs del proyecto **CancelAListOfTasks**, inicie la transición moviendo los pasos de

procesamiento para cada sitio web desde el bucle en `AccessTheWebAsync` al siguiente método asincrónico.

```
// ***Bundle the processing steps for a website into one async method.
async Task<int> ProcessURLAsync(string url, HttpClient client, CancellationToken ct)
{
    // GetAsync returns a Task<HttpResponseMessage>.
    HttpResponseMessage response = await client.GetAsync(url, ct);

    // Retrieve the website contents from the HttpResponseMessage.
    byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

    return urlContents.Length;
}
```

En `AccessTheWebAsync`, este ejemplo usa una consulta, el método ToArray y el método `WhenAny` para crear e iniciar una matriz de tareas. La aplicación de `WhenAny` a la matriz devuelve una única tarea que, cuando se espera, se evalúa como la primera tarea que llega a la finalización de la matriz de tareas.

Realice los siguientes cambios en `AccessTheWebAsync`. Los asteriscos marcan los cambios en el archivo de código.

1. Convierta en comentario o elimine el bucle.

2. Cree una consulta que, cuando se ejecute, genere una colección de tareas genéricas. Cada llamada a `ProcessURLAsync` devuelve un Task<TResult> donde `TResult` es un entero.

   ```
   // ***Create a query that, when executed, returns a collection of tasks.
   IEnumerable<Task<int>> downloadTasksQuery =
       from url in urlList select ProcessURLAsync(url, client, ct);
   ```

3. Llame a `ToArray` para ejecutar la consulta e iniciar las tareas. La aplicación del método `WhenAny` en el paso siguiente ejecutaría la consulta e iniciaría las tareas sin usar `ToArray`, pero es posible que otros métodos no lo hagan. La práctica más segura es forzar explícitamente la ejecución de la consulta.

   ```
   // ***Use ToArray to execute the query and start the download tasks.
   Task<int>[] downloadTasks = downloadTasksQuery.ToArray();
   ```

4. Llame a `WhenAny` en la colección de tareas. `WhenAny` devuelve una `Task(Of Task(Of Integer))` o `Task<Task<int>>`. Es decir, `WhenAny` devuelve una tarea que se evalúa como una única `Task(Of Integer)` o `Task<int>` cuando se espera. Esa única tarea es la primera tarea de la colección en finalizar. La tarea que finalizó primero se asigna a `firstFinishedTask`. El tipo de `firstFinishedTask` es Task<TResult>, donde `TResult` es un entero, ya que es el tipo de valor devuelto de `ProcessURLAsync`.

   ```
   // ***Call WhenAny and then await the result. The task that finishes
   // first is assigned to firstFinishedTask.
   Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);
   ```

5. En este ejemplo, solo le interesa la tarea que finaliza primero. Por lo tanto, use CancellationTokenSource.Cancel para cancelar las tareas restantes.

   ```
   // ***Cancel the rest of the downloads. You just want the first one.
   cts.Cancel();
   ```

6. Por último, espere a `firstFinishedTask` para recuperar la longitud del contenido descargado.

```
var length = await firstFinishedTask;
resultsTextBox.Text += $"\r\nLength of the downloaded website:  {length}\r\n";
```

Ejecute el programa varias veces para comprobar que finalizan primero descargas diferentes.

Ejemplo completo

El código siguiente es el archivo MainWindow.xaml.cs completo para el ejemplo. Los asteriscos marcan los elementos que se han agregado a este ejemplo.

Observe que debe agregar una referencia para System.Net.Http.

Puede descargar el proyecto de Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add a using directive and a reference for System.Net.Http.
using System.Net.Http;

// Add the following using directive.
using System.Threading;

namespace CancelAfterOneTask
{
    public partial class MainWindow : Window
    {
        // Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            // Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            resultsTextBox.Clear();

            try
            {
                await AccessTheWebAsync(cts.Token);
                resultsTextBox.Text += "\r\nDownload complete.";
            }
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownload canceled.";
            }
```

```
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownload failed.";
            }

            // Set the CancellationTokenSource to null when the download is complete.
            cts = null;
        }

        // You can still include a Cancel button if you want to.
        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        // Provide a parameter for the CancellationToken.
        async Task AccessTheWebAsync(CancellationToken ct)
        {
            HttpClient client = new HttpClient();

            // Call SetUpURLList to make a list of web addresses.
            List<string> urlList = SetUpURLList();

            // ***Comment out or delete the loop.
            //foreach (var url in urlList)
            //{
            //    // GetAsync returns a Task<HttpResponseMessage>.
            //    // Argument ct carries the message if the Cancel button is chosen.
            //    // ***Note that the Cancel button can cancel all remaining downloads.
            //    HttpResponseMessage response = await client.GetAsync(url, ct);

            //    // Retrieve the website contents from the HttpResponseMessage.
            //    byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

            //    resultsTextBox.Text +=
            //        $"\r\nLength of the downloaded string: {urlContents.Length}.\r\n";
            //}

            // ***Create a query that, when executed, returns a collection of tasks.
            IEnumerable<Task<int>> downloadTasksQuery =
                from url in urlList select ProcessURLAsync(url, client, ct);

            // ***Use ToArray to execute the query and start the download tasks.
            Task<int>[] downloadTasks = downloadTasksQuery.ToArray();

            // ***Call WhenAny and then await the result. The task that finishes
            // first is assigned to firstFinishedTask.
            Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);

            // ***Cancel the rest of the downloads. You just want the first one.
            cts.Cancel();

            // ***Await the first completed task and display the results.
            // Run the program several times to demonstrate that different
            // websites can finish first.
            var length = await firstFinishedTask;
            resultsTextBox.Text += $"\r\nLength of the downloaded website:  {length}\r\n";
        }

        // ***Bundle the processing steps for a website into one async method.
        async Task<int> ProcessURLAsync(string url, HttpClient client, CancellationToken ct)
        {
            // GetAsync returns a Task<HttpResponseMessage>.
            HttpResponseMessage response = await client.GetAsync(url, ct);

            // Retrieve the website contents from the HttpResponseMessage.
```

```
            byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

            return urlContents.Length;
        }

        // Add a method that creates a list of web addresses.
        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/hh290138.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/dd470362.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }
    }
    // Sample output:

    // Length of the downloaded website:  158856

    // Download complete.
}
```

Vea también

- WhenAny
- Fine-Tuning Your Async Application (C#) (Ajuste de la aplicación asincrónica [C#])
- Asynchronous Programming with async and await (C#) (Programación asincrónica con async y await (C#))
- Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación)

Iniciar varias tareas asincrónicas y procesarlas a medida que se completan (C#)

18/03/2020 • 8 minutes to read • Edit Online

Si usa Task.WhenAny, puede iniciar varias tareas a la vez y procesarlas una por una a medida que se completen, en lugar de procesarlas en el orden en el que se han iniciado.

En el siguiente ejemplo se usa una consulta para crear una colección de tareas. Cada tarea descarga el contenido de un sitio web especificado. En cada iteración de un bucle while, una llamada awaited a `WhenAny` devuelve la tarea en la colección de tareas que termine primero su descarga. Esa tarea se quita de la colección y se procesa. El bucle se repite hasta que la colección no contiene más tareas.

> **NOTE**
>
> Para ejecutar los ejemplos, debe tener Visual Studio 2012 o posterior, y .NET Framework 4.5 o posterior, instalado en el equipo.

Descargue una solución de ejemplo

Puede descargar el proyecto completo de Windows Presentation Foundation (WPF) en Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]) y después seguir estos pasos.

> **TIP**
>
> Si no quiere descargar el proyecto, en su lugar, puede revisar el archivo *MainWindow.xaml.cs* al final de este tema.

1. Extraiga los archivos que ha descargado del archivo *.zip* y, después, inicie Visual Studio.
2. En la barra de menús, elija **Archivo** > **Abrir** > **Proyecto/Solución**.
3. En el cuadro de diálogo **Abrir proyecto**, abra la carpeta que contiene el código de ejemplo que descargó y, después, abra el archivo de solución (*.sln)* para *AsyncFineTuningCS/AsyncFineTuningVB*.
4. En el **Explorador de soluciones**, abra el menú contextual del proyecto **ProcessTasksAsTheyFinish** y, después, pulse **Establecer como proyecto de inicio**.
5. Presione la tecla `F5` para ejecutar el programa con depuración, o bien presione las teclas `Ctrl+F5` para ejecutar el programa sin depurarlo.
6. Ejecute el proyecto varias veces para comprobar que las longitudes que se han descargado no aparecen siempre en el mismo orden.

Crear el programa usted mismo

Este ejemplo se agrega al código desarrollado en Cancelar las tareas asincrónicas restantes cuando se completa una (C#) y usa la misma interfaz de usuario.

Para generar el ejemplo personalmente, paso a paso, siga las instrucciones de la sección Descargar el ejemplo, pero elija **CancelAfterOneTask** como el proyecto de inicio. Agregue los cambios de este tema al método `AccessTheWebAsync` de ese proyecto. Los cambios se marcan con asteriscos.

El proyecto **CancelAfterOneTask** ya incluye una consulta que, cuando se ejecuta, crea una colección de tareas. Cada llamada a `ProcessURLAsync` en el siguiente código devuelve un objeto Task<TResult>, donde `TResult` es un entero:

```
IEnumerable<Task<int>> downloadTasksQuery = from url in urlList select ProcessURL(url, client, ct);
```

En el archivo *MainWindow.xaml.cs* del proyecto, realice los siguientes cambios en el método `AccessTheWebAsync`:

- Ejecute la consulta aplicando Enumerable.ToList en lugar de ToArray.

  ```
  List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
  ```

- Agregue un bucle `while` que realice los pasos siguientes para cada tarea de la colección:

  1. Espera una llamada a `WhenAny` para identificar la primera tarea en la colección para finalizar su descarga.

     ```
     Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);
     ```

  2. Quita la tarea de la colección.

     ```
     downloadTasks.Remove(firstFinishedTask);
     ```

  3. Espera `firstFinishedTask`, que se devuelve mediante una llamada a `ProcessURLAsync`. La variable `firstFinishedTask` es un Task<TResult> donde `TReturn` es un entero. La tarea ya está completa, pero la espera para recuperar la longitud del sitio web descargado, como se muestra en el ejemplo siguiente. Si se produce un error en la tarea, `await` iniciará la primera excepción secundaria almacenada en `AggregateException`, a diferencia de leer la propiedad `Result` que iniciaría `AggregateException`.

     ```
     int length = await firstFinishedTask;
     resultsTextBox.Text += $"\r\nLength of the download:  {length}";
     ```

Ejecute el programa varias veces para comprobar que las longitudes que se han descargado no aparecen siempre en el mismo orden.

Caution

Puede usar `WhenAny` en un bucle, como se describe en el ejemplo, para solucionar problemas que implican un número reducido de tareas. Sin embargo, otros enfoques son más eficaces si hay que procesar un gran número de tareas. Para más información y ejemplos, vea Processing Tasks as they complete (Procesar tareas a medida que se completan).

Ejemplo completo

El código siguiente es el texto completo del archivo *MainWindow.xaml.cs* para el ejemplo. Los asteriscos marcan los elementos que se han agregado a este ejemplo. Además, observe que debe agregar una referencia para System.Net.Http.

Puede descargar el proyecto de Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación [C# y Visual Basic]).

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add a using directive and a reference for System.Net.Http.
using System.Net.Http;

// Add the following using directive.
using System.Threading;

namespace ProcessTasksAsTheyFinish
{
    public partial class MainWindow : Window
    {
        // Declare a System.Threading.CancellationTokenSource.
        CancellationTokenSource cts;

        public MainWindow()
        {
            InitializeComponent();
        }

        private async void startButton_Click(object sender, RoutedEventArgs e)
        {
            resultsTextBox.Clear();

            // Instantiate the CancellationTokenSource.
            cts = new CancellationTokenSource();

            try
            {
                await AccessTheWebAsync(cts.Token);
                resultsTextBox.Text += "\r\nDownloads complete.";
            }
            catch (OperationCanceledException)
            {
                resultsTextBox.Text += "\r\nDownloads canceled.\r\n";
            }
            catch (Exception)
            {
                resultsTextBox.Text += "\r\nDownloads failed.\r\n";
            }

            cts = null;
        }

        private void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            if (cts != null)
            {
                cts.Cancel();
            }
        }

        async Task AccessTheWebAsync(CancellationToken ct)
        {
            HttpClient client = new HttpClient();

            // Make a list of web addresses.
```

```
            List<string> urlList = SetUpURLList();

            // ***Create a query that, when executed, returns a collection of tasks.
            IEnumerable<Task<int>> downloadTasksQuery =
                from url in urlList select ProcessURL(url, client, ct);

            // ***Use ToList to execute the query and start the tasks.
            List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

            // ***Add a loop to process the tasks one at a time until none remain.
            while (downloadTasks.Count > 0)
            {
                    // Identify the first task that completes.
                    Task<int> firstFinishedTask = await Task.WhenAny(downloadTasks);

                    // ***Remove the selected task from the list so that you don't
                    // process it more than once.
                    downloadTasks.Remove(firstFinishedTask);

                    // Await the completed task.
                    int length = await firstFinishedTask;
                    resultsTextBox.Text += $"\r\nLength of the download:  {length}";
            }
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/windows/apps/br211380.aspx",
                "https://msdn.microsoft.com/library/hh290136.aspx",
                "https://msdn.microsoft.com/library/dd470362.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }

        async Task<int> ProcessURL(string url, HttpClient client, CancellationToken ct)
        {
            // GetAsync returns a Task<HttpResponseMessage>.
            HttpResponseMessage response = await client.GetAsync(url, ct);

            // Retrieve the website contents from the HttpResponseMessage.
            byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

            return urlContents.Length;
        }
    }
}

// Sample Output:

// Length of the download:  226093
// Length of the download:  412588
// Length of the download:  175490
// Length of the download:  204890
// Length of the download:  158855
// Length of the download:  145790
// Length of the download:  44908
// Downloads complete.
```

Vea también

- WhenAny
- Fine-Tuning Your Async Application (C#) (Ajuste de la aplicación asincrónica [C#])
- Asynchronous Programming with async and await (C#) (Programación asincrónica con async y await (C#))
- Async Sample: Fine Tuning Your Application (Ejemplo asincrónico: Ajustar la aplicación)

Controlar la reentrada en aplicaciones asincrónicas (C#)

29/06/2020 • 29 minutes to read • Edit Online

Cuando se incluye código asincrónico en una aplicación, hay que tener en cuenta (y posiblemente evitar) la reentrada, que significa volver a especificar una operación asincrónica antes de que finalice. Si no se identifican ni controlan las posibilidades de reentrada, pueden producirse resultados inesperados.

En este tema

- Reconocer la reentrada
- Controlar la reentrada
  - Deshabilitar el botón de inicio
  - Cancelar y reiniciar la operación
  - Ejecutar varias operaciones y poner en cola la salida
- Revisión y ejecución de la aplicación de ejemplo

> **NOTE**
> Para ejecutar el ejemplo, debe tener instalado en el equipo Visual Studio 2012 o posterior y .NET Framework 4.5 o posterior.

> **NOTE**
> La versión 1.2 de Seguridad de la capa de transporte (TLS) es ahora la versión mínima que se usará en el desarrollo de la aplicación. Si la aplicación tiene como destino una versión de .NET Framework anterior a la 4.7, consulte el artículo siguiente para obtener Prácticas recomendadas de Seguridad de la capa de transporte (TLS) con .NET Framework.

Reconocer la reentrada

En el ejemplo de este tema, los usuarios hacen clic en un botón **Start** (Iniciar) para iniciar una aplicación asincrónica que descarga una serie de sitios web y calcula el número total de bytes que se descargan. Una versión sincrónica del ejemplo respondería de la misma forma independientemente de cuántas veces un usuario elija el botón porque, tras la primera vez, el subproceso de UI omite esos eventos hasta que finaliza la ejecución de la aplicación. Sin embargo, en una aplicación asincrónica, el subproceso de UI continúa respondiendo y podría volver a introducir la operación asincrónica antes de que finalice.

En el ejemplo siguiente se muestra la salida esperada si el usuario hace clic en el botón **Start** una sola vez. Aparece una lista de los sitios web descargados con el tamaño, en bytes, de cada sitio. El número total de bytes aparece al final.

```
1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
3. msdn.microsoft.com/library/jj155761.aspx                29019
4. msdn.microsoft.com/library/hh290140.aspx               117152
5. msdn.microsoft.com/library/hh524395.aspx                68959
6. msdn.microsoft.com/library/ms404677.aspx               197325
7. msdn.microsoft.com                                            42972
8. msdn.microsoft.com/library/ff730837.aspx               146159

TOTAL bytes returned:  890591
```

Sin embargo, si el usuario elige el botón más de una vez, el controlador de eventos se invoca repetidamente y el proceso de descarga se vuelve a introducir cada vez. Como resultado, se ejecutan varias operaciones asincrónicas al mismo tiempo, la salida intercala los resultados y el número total de bytes es confuso.

```
1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
3. msdn.microsoft.com/library/jj155761.aspx                29019
4. msdn.microsoft.com/library/hh290140.aspx               117152
5. msdn.microsoft.com/library/hh524395.aspx                68959
1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
6. msdn.microsoft.com/library/ms404677.aspx               197325
3. msdn.microsoft.com/library/jj155761.aspx                29019
7. msdn.microsoft.com                                            42972
4. msdn.microsoft.com/library/hh290140.aspx               117152
8. msdn.microsoft.com/library/ff730837.aspx               146159

TOTAL bytes returned:  890591

5. msdn.microsoft.com/library/hh524395.aspx                68959
1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
6. msdn.microsoft.com/library/ms404677.aspx               197325
3. msdn.microsoft.com/library/jj155761.aspx                29019
4. msdn.microsoft.com/library/hh290140.aspx               117152
7. msdn.microsoft.com                                            42972
5. msdn.microsoft.com/library/hh524395.aspx                68959
8. msdn.microsoft.com/library/ff730837.aspx               146159

TOTAL bytes returned:  890591

6. msdn.microsoft.com/library/ms404677.aspx               197325
7. msdn.microsoft.com                                            42972
8. msdn.microsoft.com/library/ff730837.aspx               146159

TOTAL bytes returned:  890591
```

Al final de este tema puede revisar el código que genera este resultado. Si quiere experimentar con el código, descargue la solución en el equipo local y ejecute el proyecto WebsiteDownload o use el código que aparece al final de este tema para crear su propio proyecto. Para más información e instrucciones, vea Revisión y ejecución de la aplicación de ejemplo.

Controlar la reentrada

La reentrada se puede controlar de varias maneras en función de lo que se desee de la aplicación. Este tema presenta los siguientes ejemplos:

- Deshabilitar el botón de inicio

  Deshabilite el botón **Start** (Iniciar) mientras se ejecuta la operación de modo que el usuario no pueda

interrumpirla.

- Cancelar y reiniciar la operación

  Cancele cualquier operación que se esté ejecutando cuando el usuario haga clic de nuevo en el botón **Start** y, después, deje que continúe la última operación solicitada.

- Ejecutar varias operaciones y poner en cola la salida

  Permita que todas las operaciones solicitadas se ejecuten de forma asincrónica, pero coordine la presentación de salida para que los resultados de cada operación aparecen juntos y en orden.

Deshabilitar el botón de inicio

Puede bloquear el botón **Start** mientras se ejecuta una operación si lo deshabilita en la parte superior del controlador de eventos `StartButton_Click`. A continuación, cuando finalice la operación, puede habilitar de nuevo el botón desde un bloque `finally` de modo que los usuarios puedan volver a ejecutar la aplicación.

Para configurar este escenario, haga los cambios siguientes en el código básico que se proporciona en Revisión y ejecución de la aplicación de ejemplo. También puede descargar la aplicación finalizada de Async Samples: Reentrancy in .NET Desktop Apps (Ejemplos asincrónicos: reentrada en aplicaciones de escritorio de .NET). El nombre del proyecto es DisableStartButton.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // This line is commented out to make the results clearer in the output.
    //ResultsTextBox.Text = "";

    // ***Disable the Start button until the downloads are complete.
    StartButton.IsEnabled = false;

    try
    {
        await AccessTheWebAsync();
    }
    catch (Exception)
    {
        ResultsTextBox.Text += "\r\nDownloads failed.";
    }
    // ***Enable the Start button in case you want to run the program again.
    finally
    {
        StartButton.IsEnabled = true;
    }
}
```

Como resultado de los cambios, el botón no responde mientras `AccessTheWebAsync` está descargando los sitios web, por lo que no se puede volver a introducir el proceso.

Cancelar y reiniciar la operación

En lugar de deshabilitar el botón **Start**, puede mantenerlo activo y, si el usuario vuelve a seleccionarlo, cancelar la operación que ya se está ejecutando y permitir que la última operación iniciada continúe.

Para más información sobre la cancelación, vea Fine-Tuning Your Async Application (C#) (Ajustar la aplicación asincrónica [C#]).

Para configurar este escenario, haga los cambios siguientes en el código básico que se proporciona en Revisión y ejecución de la aplicación de ejemplo. También puede descargar la aplicación finalizada de Async Samples: Reentrancy in .NET Desktop Apps (Ejemplos asincrónicos: reentrada en aplicaciones de escritorio de .NET). El nombre del proyecto es CancelAndRestart.

1. Declare una variable de CancellationTokenSource, `cts`, que esté en el ámbito de todos los métodos.

```
public partial class MainWindow : Window   // Or class MainPage
{
    // *** Declare a System.Threading.CancellationTokenSource.
    CancellationTokenSource cts;
```

2. En `StartButton_Click`, determine si una operación ya está en curso. Si el valor de `cts` es NULL, ya no hay ninguna operación activa. Si el valor no es null, se cancela la operación que se esté ejecutando.

```
// *** If a download process is already underway, cancel it.
if (cts != null)
{
    cts.Cancel();
}
```

3. Establezca `cts` en un valor diferente que represente el proceso actual.

```
// *** Now set cts to a new value that you can use to cancel the current process
// if the button is chosen again.
CancellationTokenSource newCTS = new CancellationTokenSource();
cts = newCTS;
```

4. Al final de `StartButton_Click`, el proceso actual está completo, así que vuelva a establecer el valor de `cts` en null.

```
// *** When the process is complete, signal that another process can begin.
if (cts == newCTS)
    cts = null;
```

El código siguiente muestra todos los cambios en `StartButton_Click`. Las adiciones se marcan con asteriscos.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // This line is commented out to make the results clearer in the output.
    //ResultsTextBox.Clear();

    // *** If a download process is already underway, cancel it.
    if (cts != null)
    {
        cts.Cancel();
    }

    // *** Now set cts to cancel the current process if the button is chosen again.
    CancellationTokenSource newCTS = new CancellationTokenSource();
    cts = newCTS;

    try
    {
        // ***Send cts.Token to carry the message if there is a cancellation request.
        await AccessTheWebAsync(cts.Token);

    }
    // *** Catch cancellations separately.
    catch (OperationCanceledException)
    {
        ResultsTextBox.Text += "\r\nDownloads canceled.\r\n";
    }
    catch (Exception)
    {
        ResultsTextBox.Text += "\r\nDownloads failed.\r\n";
    }
    // *** When the process is complete, signal that another process can proceed.
    if (cts == newCTS)
        cts = null;
}
```

En `AccessTheWebAsync`, realice los siguientes cambios.

- Agregue un parámetro para aceptar el token de cancelación de `StartButton_Click`.
- Use el método GetAsync para descargar los sitios web porque `GetAsync` acepta un argumento CancellationToken.
- Antes de llamar a `DisplayResults` para mostrar los resultados de los sitios web descargados, revise `ct` para comprobar que no se ha cancelado la operación actual.

El código siguiente muestra estos cambios marcados con asteriscos.

```
// *** Provide a parameter for the CancellationToken from StartButton_Click.
async Task AccessTheWebAsync(CancellationToken ct)
{
    // Declare an HttpClient object.
    HttpClient client = new HttpClient();

    // Make a list of web addresses.
    List<string> urlList = SetUpURLList();

    var total = 0;
    var position = 0;

    foreach (var url in urlList)
    {
        // *** Use the HttpClient.GetAsync method because it accepts a
        // cancellation token.
        HttpResponseMessage response = await client.GetAsync(url, ct);

        // *** Retrieve the website contents from the HttpResponseMessage.
        byte[] urlContents = await response.Content.ReadAsByteArrayAsync();

        // *** Check for cancellations before displaying information about the
        // latest site.
        ct.ThrowIfCancellationRequested();

        DisplayResults(url, urlContents, ++position);

        // Update the total.
        total += urlContents.Length;
    }

    // Display the total count for all of the websites.
    ResultsTextBox.Text +=
        $"\r\n\r\nTOTAL bytes returned:  {total}\r\n";
}
```

Si hace clic varias veces en el botón **Start** mientras se ejecuta esta aplicación, debería producir resultados similares a la salida siguiente.

```
1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
3. msdn.microsoft.com/library/jj155761.aspx                29019
4. msdn.microsoft.com/library/hh290140.aspx               122505
5. msdn.microsoft.com/library/hh524395.aspx                68959
6. msdn.microsoft.com/library/ms404677.aspx               197325
Download canceled.

1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
3. msdn.microsoft.com/library/jj155761.aspx                29019
Download canceled.

1. msdn.microsoft.com/library/hh191443.aspx                83732
2. msdn.microsoft.com/library/aa578028.aspx               205273
3. msdn.microsoft.com/library/jj155761.aspx                29019
4. msdn.microsoft.com/library/hh290140.aspx               117152
5. msdn.microsoft.com/library/hh524395.aspx                68959
6. msdn.microsoft.com/library/ms404677.aspx               197325
7. msdn.microsoft.com                                            42972
8. msdn.microsoft.com/library/ff730837.aspx               146159

TOTAL bytes returned:  890591
```

Para eliminar las listas parciales, quite el comentario de la primera línea de código en `StartButton_Click` para

borrar el cuadro de texto cada vez que el usuario reinicie la operación.

Ejecutar varias operaciones y poner en cola el resultado

Este tercer ejemplo es el más complicado porque la aplicación inicia otra operación asincrónica cada vez que el usuario selecciona el botón **Start** y todas las operaciones se ejecutan hasta completarse. Todas las operaciones solicitadas descargan los sitios web de la lista de forma asincrónica, pero la salida de las operaciones se presenta de manera secuencial. Es decir, la actividad de descarga real se intercala, según se muestra en la salida de Reconocer la reentrada, pero la lista de resultados de cada grupo se presenta por separado.

Las operaciones comparten una Task global, `pendingWork`, que actúa de equipo selector para el proceso de visualización.

Para configurar este escenario, haga los cambios siguientes en el código básico que se proporciona en Revisión y ejecución de la aplicación de ejemplo. También puede descargar la aplicación finalizada de Async Samples: Reentrancy in .NET Desktop Apps (Ejemplos asincrónicos: reentrada en aplicaciones de escritorio de .NET). El nombre del proyecto es QueueResults.

En la salida siguiente se muestra el resultado cuando el usuario selecciona el botón **Start** una sola vez. La etiqueta de letra A indica que el resultado se corresponde a la primera vez que se selecciona el botón **Start**. Los números muestran el orden de las direcciones URL en la lista de destinos de descarga.

```
#Starting group A.
#Task assigned for group A.

A-1. msdn.microsoft.com/library/hh191443.aspx                87389
A-2. msdn.microsoft.com/library/aa578028.aspx               209858
A-3. msdn.microsoft.com/library/jj155761.aspx                30870
A-4. msdn.microsoft.com/library/hh290140.aspx               119027
A-5. msdn.microsoft.com/library/hh524395.aspx                71260
A-6. msdn.microsoft.com/library/ms404677.aspx               199186
A-7. msdn.microsoft.com                                            53266
A-8. msdn.microsoft.com/library/ff730837.aspx               148020

TOTAL bytes returned:  918876

#Group A is complete.
```

Si el usuario hace clic tres veces en el botón **Start**, la aplicación genera una salida similar a las líneas siguientes. Las líneas de información que comienzan con una almohadilla (#) siguen el progreso de la aplicación.

```
#Starting group A.
#Task assigned for group A.

A-1. msdn.microsoft.com/library/hh191443.aspx                87389
A-2. msdn.microsoft.com/library/aa578028.aspx               207089
A-3. msdn.microsoft.com/library/jj155761.aspx                30870
A-4. msdn.microsoft.com/library/hh290140.aspx               119027
A-5. msdn.microsoft.com/library/hh524395.aspx                71259
A-6. msdn.microsoft.com/library/ms404677.aspx               199185

#Starting group B.
#Task assigned for group B.

A-7. msdn.microsoft.com                                            53266

#Starting group C.
#Task assigned for group C.

A-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  916095

B-1. msdn.microsoft.com/library/hh191443.aspx                87389
B-2. msdn.microsoft.com/library/aa578028.aspx               207089
B-3. msdn.microsoft.com/library/jj155761.aspx                30870
B-4. msdn.microsoft.com/library/hh290140.aspx               119027
B-5. msdn.microsoft.com/library/hh524395.aspx                71260
B-6. msdn.microsoft.com/library/ms404677.aspx               199186

#Group A is complete.

B-7. msdn.microsoft.com                                            53266
B-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  916097

C-1. msdn.microsoft.com/library/hh191443.aspx                87389
C-2. msdn.microsoft.com/library/aa578028.aspx               207089

#Group B is complete.

C-3. msdn.microsoft.com/library/jj155761.aspx                30870
C-4. msdn.microsoft.com/library/hh290140.aspx               119027
C-5. msdn.microsoft.com/library/hh524395.aspx                72765
C-6. msdn.microsoft.com/library/ms404677.aspx               199186
C-7. msdn.microsoft.com                                            56190
C-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  920526

#Group C is complete.
```

Los grupos B y C se inician antes de finalizar el grupo A, pero la salida de cada grupo aparece por separado. Primero aparece toda la salida del grupo A, seguida de la salida del grupo B y después la del grupo C. La aplicación siempre muestra los grupos en orden y, en cada grupo, muestra la información sobre los sitios web individuales en el orden en que las direcciones URL aparecen en la lista de direcciones URL.

Sin embargo, no es posible predecir el orden en que se producen las descargas. Después de iniciarse varios grupos, las tareas de descarga que generan están activas. No se puede dar por sentado que A-1 se descargará antes que B-1, ni que A-1 se descargará antes que A-2.

Definiciones globales

El código de ejemplo contiene las dos declaraciones globales siguientes que están visibles en todos los métodos.

```
public partial class MainWindow : Window  // Class MainPage in Windows Store app.
{
    // ***Declare the following variables where all methods can access them.
    private Task pendingWork = null;
    private char group = (char)('A' - 1);
```

La variable de `Task`, `pendingWork`, supervisa el proceso de presentación e impide que un grupo interrumpa la operación de presentación de otro grupo. La variable de caracteres, `group`, etiqueta la salida de diferentes grupos para comprobar que los resultados aparecen en el orden esperado.

El controlador de eventos Click

El controlador de eventos, `StartButton_Click`, incrementa la letra del grupo cada vez que el usuario selecciona el botón **Start**. A continuación, el controlador llama a `AccessTheWebAsync` para ejecutar la operación de descarga.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ***Verify that each group's results are displayed together, and that
    // the groups display in order, by marking each group with a letter.
    group = (char)(group + 1);
    ResultsTextBox.Text += $"\r\n\r\n#Starting group {group}.";

    try
    {
        // *** Pass the group value to AccessTheWebAsync.
        char finishedGroup = await AccessTheWebAsync(group);

        // The following line verifies a successful return from the download and
        // display procedures.
        ResultsTextBox.Text += $"\r\n\r\n#Group {finishedGroup} is complete.\r\n";
    }
    catch (Exception)
    {
        ResultsTextBox.Text += "\r\nDownloads failed.";
    }
}
```

El método AccessTheWebAsync

En este ejemplo se divide `AccessTheWebAsync` en dos métodos. El primer método, `AccessTheWebAsync`, inicia todas las tareas de descarga de un grupo y configura `pendingWork` para controlar el proceso de visualización. El método usa una consulta de Language Integrated Query (consulta LINQ) y ToArray para iniciar todas las tareas de descarga al mismo tiempo.

A continuación, `AccessTheWebAsync` llama a `FinishOneGroupAsync` para esperar la finalización de todas las descargas y mostrar su duración.

`FinishOneGroupAsync` devuelve una tarea que se asigna a `pendingWork` en `AccessTheWebAsync`. Ese valor evita que otra operación interrumpa la tarea antes de que finalice.

```
private async Task<char> AccessTheWebAsync(char grp)
{
    HttpClient client = new HttpClient();

    // Make a list of the web addresses to download.
    List<string> urlList = SetUpURLList();

    // ***Kick off the downloads. The application of ToArray activates all the download tasks.
    Task<byte[]>[] getContentTasks = urlList.Select(url => client.GetByteArrayAsync(url)).ToArray();

    // ***Call the method that awaits the downloads and displays the results.
    // Assign the Task that FinishOneGroupAsync returns to the gatekeeper task, pendingWork.
    pendingWork = FinishOneGroupAsync(urlList, getContentTasks, grp);

    ResultsTextBox.Text += $"\r\n#Task assigned for group {grp}. Download tasks are active.\r\n";

    // ***This task is complete when a group has finished downloading and displaying.
    await pendingWork;

    // You can do other work here or just return.
    return grp;
}
```

El método FinishOneGroupAsync

Este método recorre las tareas de descarga de un grupo, espera por cada una de ellas, muestra la longitud del sitio web descargado y agrega la longitud total.

La primera instrucción de `FinishOneGroupAsync` usa `pendingWork` para asegurarse de que la entrada al método no interfiere con una operación que ya está en el proceso de visualización o que ya está esperando. Si tal operación está en curso, la operación introducida debe esperar su turno.

```
private async Task FinishOneGroupAsync(List<string> urls, Task<byte[]>[] contentTasks, char grp)
{
    // ***Wait for the previous group to finish displaying results.
    if (pendingWork != null) await pendingWork;

    int total = 0;

    // contentTasks is the array of Tasks that was created in AccessTheWebAsync.
    for (int i = 0; i < contentTasks.Length; i++)
    {
        // Await the download of a particular URL, and then display the URL and
        // its length.
        byte[] content = await contentTasks[i];
        DisplayResults(urls[i], content, i, grp);
        total += content.Length;
    }

    // Display the total count for all of the websites.
    ResultsTextBox.Text +=
        $"\r\n\r\nTOTAL bytes returned:  {total}\r\n";
}
```

Puntos de interés

Las líneas de información que comienzan con un signo de almohadilla (#) en la salida aclaran cómo funciona este ejemplo.

La salida muestra los siguientes patrones.

- Un grupo puede iniciarse mientras un grupo anterior muestra su salida, pero la visualización del grupo anterior no se ve interrumpida.

```
#Starting group A.
#Task assigned for group A. Download tasks are active.

A-1. msdn.microsoft.com/library/hh191443.aspx                87389
A-2. msdn.microsoft.com/library/aa578028.aspx               207089
A-3. msdn.microsoft.com/library/jj155761.aspx                30870
A-4. msdn.microsoft.com/library/hh290140.aspx               119037
A-5. msdn.microsoft.com/library/hh524395.aspx                71260

#Starting group B.
#Task assigned for group B. Download tasks are active.

A-6. msdn.microsoft.com/library/ms404677.aspx               199186
A-7. msdn.microsoft.com                                            53078
A-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  915919

B-1. msdn.microsoft.com/library/hh191443.aspx                87388
B-2. msdn.microsoft.com/library/aa578028.aspx               207089
B-3. msdn.microsoft.com/library/jj155761.aspx                30870

#Group A is complete.

B-4. msdn.microsoft.com/library/hh290140.aspx               119027
B-5. msdn.microsoft.com/library/hh524395.aspx                71260
B-6. msdn.microsoft.com/library/ms404677.aspx               199186
B-7. msdn.microsoft.com                                            53078
B-8. msdn.microsoft.com/library/ff730837.aspx               148010

TOTAL bytes returned:  915908
```

- La tarea `pendingWork` es NULL al inicio de `FinishOneGroupAsync` solo para el grupo A, que fue el primero en empezar. El grupo A todavía no ha completado una expresión await cuando alcanza `FinishOneGroupAsync`. Por lo tanto, el control no se ha devuelto a `AccessTheWebAsync`, y la primera asignación a `pendingWork` no se ha producido.

- Las dos líneas siguientes siempre aparecen juntas en la salida. El código no se interrumpa nunca entre el inicio de la operación de un grupo en de `StartButton_Click` y la asignación de una tarea del grupo a `pendingWork`.

  ```
  #Starting group B.
  #Task assigned for group B. Download tasks are active.
  ```

  Una vez que un grupo introduce `StartButton_Click`, la operación no completa una expresión await hasta que la operación introduce `FinishOneGroupAsync`. Por lo tanto, ninguna otra operación puede lograr el control durante ese segmento de código.

Revisión y ejecución de la aplicación de ejemplo

Para entender mejor la aplicación de ejemplo, puede descargarla, compilarla usted mismo o revisar el código al final de este tema sin necesidad de implementar la aplicación.

> **NOTE**
>
> Para ejecutar el ejemplo como aplicación de escritorio de Windows Presentation Foundation (WPF), debe tener instalado en el equipo Visual Studio 2012 o posterior, y .NET Framework 4.5 o posterior.

Descargar la aplicación

1. Descargue el archivo comprimido de Async Samples: Reentrancy in .NET Desktop Apps (Ejemplos asincrónicos: reentrada en aplicaciones de escritorio de .NET).

2. Descomprima el archivo descargado y, a continuación, inicie Visual Studio.

3. En la barra de menús, elija **Archivo**, **Abrir**, **Proyecto o solución**.

4. Navegue hasta la carpeta que contiene el código de ejemplo descomprimido y, a continuación, abra el archivo de solución (.sln).

5. En el **Explorador de soluciones**, abra el menú contextual del proyecto que quiere ejecutar y, después, elija **Establecer como proyecto de inicio**.

6. Elija las teclas CTRL+F5 para compilar y ejecutar el proyecto.

Compilar la aplicación

La sección siguiente proporciona el código para compilar el ejemplo como una aplicación de WPF.

Para compilar una aplicación WPF

1. Inicie Visual Studio.

2. En la barra de menús, elija **Archivo**, **Nuevo**, **Proyecto**.

   Aparece el cuadro de diálogo **Nuevo proyecto** .

3. En el panel **Plantillas instaladas**, expanda **Visual C#** y después **Windows**.

4. En la lista de tipos de proyecto, seleccione **Aplicación WPF**.

5. Asigne un nombre al proyecto `WebsiteDownloadWPF`, elija la versión 4.6 de .NET Framework o una posterior y, después, haga clic en el botón **Aceptar**.

   El proyecto nuevo aparece en el **Explorador de soluciones**.

6. En el Editor de código de Visual Studio, elija la pestaña **MainWindow.xaml** .

   Si la pestaña no está visible, abra el menú contextual de MainWindow.xaml en el **Explorador de soluciones**y elija **Ver código**.

7. En la vista **XAML** de MainWindow.xaml, reemplace el código por el código siguiente.

```
<Window x:Class="WebsiteDownloadWPF.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WebsiteDownloadWPF"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Width="517" Height="360">
        <Button x:Name="StartButton" Content="Start" HorizontalAlignment="Left" Margin="-1,0,0,0"
VerticalAlignment="Top" Click="StartButton_Click" Height="53" Background="#FFA89B9B" FontSize="36"
Width="518"  />
        <TextBox x:Name="ResultsTextBox" HorizontalAlignment="Left" Margin="-1,53,0,-36"
TextWrapping="Wrap" VerticalAlignment="Top" Height="343" FontSize="10"
ScrollViewer.VerticalScrollBarVisibility="Visible" Width="518" FontFamily="Lucida Console" />
    </Grid>
</Window>
```

   En la vista **Diseño** de MainWindow.xaml aparece una ventana simple que contiene un cuadro de texto y un botón.

8. En el **Explorador de soluciones**, haga clic con el botón derecho en **Referencias** y seleccione **Agregar referencia**.

   Agregue una referencia para System.Net.Http, si aún no está seleccionada.

9. En el **Explorador de soluciones**, abra el menú contextual de MainWindow.xaml.cs y después elija **Ver código**.

10. Reemplace el código del archivo MainWindow.xaml.cs por el código siguiente.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

// Add the following using directives, and add a reference for System.Net.Http.
using System.Net.Http;
using System.Threading;

namespace WebsiteDownloadWPF
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            System.Net.ServicePointManager.SecurityProtocol |= System.Net.SecurityProtocolType.Tls12;
            InitializeComponent();
        }

        private async void StartButton_Click(object sender, RoutedEventArgs e)
        {
            // This line is commented out to make the results clearer in the output.
            //ResultsTextBox.Text = "";

            try
            {
                await AccessTheWebAsync();
            }
            catch (Exception)
            {
                ResultsTextBox.Text += "\r\nDownloads failed.";
            }
        }

        private async Task AccessTheWebAsync()
        {
            // Declare an HttpClient object.
            HttpClient client = new HttpClient();

            // Make a list of web addresses.
            List<string> urlList = SetUpURLList();

            var total = 0;
            var position = 0;

            foreach (var url in urlList)
            {
```

```
                // GetByteArrayAsync returns a task. At completion, the task
                // produces a byte array.
                byte[] urlContents = await client.GetByteArrayAsync(url);

                DisplayResults(url, urlContents, ++position);

                // Update the total.
                total += urlContents.Length;
            }

            // Display the total count for all of the websites.
            ResultsTextBox.Text +=
                $"\r\n\r\nTOTAL bytes returned:  {total}\r\n";
        }

        private List<string> SetUpURLList()
        {
            List<string> urls = new List<string>
            {
                "https://msdn.microsoft.com/library/hh191443.aspx",
                "https://msdn.microsoft.com/library/aa578028.aspx",
                "https://msdn.microsoft.com/library/jj155761.aspx",
                "https://msdn.microsoft.com/library/hh290140.aspx",
                "https://msdn.microsoft.com/library/hh524395.aspx",
                "https://msdn.microsoft.com/library/ms404677.aspx",
                "https://msdn.microsoft.com",
                "https://msdn.microsoft.com/library/ff730837.aspx"
            };
            return urls;
        }

        private void DisplayResults(string url, byte[] content, int pos)
        {
            // Display the length of each website. The string format is designed
            // to be used with a monospaced font, such as Lucida Console or
            // Global Monospace.

            // Strip off the "https://".
            var displayURL = url.Replace("https://", "");
            // Display position in the URL list, the URL, and the number of bytes.
            ResultsTextBox.Text += $"\n{pos}. {displayURL,-58} {content.Length,8}";
        }
    }
}
```

11. Presiones las teclas CTRL+F5 para ejecutar el programa y luego haga clic varias veces en el botón **Start**.

12. Realice los cambios de Deshabilitar el botón de inicio, Cancelar y reiniciar la operación o Ejecutar varias operaciones y poner en cola el resultado para controlar la reentrada.

Vea también

- Tutorial: Acceso a web usando Async y Await (C#)
- Programación asincrónica con Async y Await (C#)

Usar Async en acceso a archivos (C#)

29/06/2020 • 9 minutes to read • Edit Online

Puede usar la característica async para acceder a archivos. Con la característica async, se puede llamar a métodos asincrónicos sin usar devoluciones de llamada ni dividir el código en varios métodos o expresiones lambda. Para convertir código sincrónico en asincrónico, basta con llamar a un método asincrónico y no a un método sincrónico y agregar algunas palabras clave al código.

Podrían considerarse los siguientes motivos para agregar asincronía a las llamadas de acceso a archivos:

- La asincronía hace que las aplicaciones de interfaz de usuario tengan mayor capacidad de respuesta porque el subproceso de interfaz de usuario que inicia la operación puede realizar otro trabajo. Si el subproceso de interfaz de usuario debe ejecutar código que tarda mucho tiempo (por ejemplo, más de 50 milisegundos), puede inmovilizar la interfaz de usuario hasta que la E/S se complete y el subproceso de interfaz de usuario pueda volver a procesar la entrada de teclado y de mouse y otros eventos.
- La asincronía mejora la escalabilidad de ASP.NET y otras aplicaciones basadas en servidor reduciendo la necesidad de subproceso. Si la aplicación usa un subproceso dedicado por respuesta y se procesa un millar de solicitudes simultáneamente, se necesitan mil subprocesos. Las operaciones asincrónicas no suelen necesitar un subproceso durante la espera. Usan el subproceso existente de finalización de E/S brevemente al final.
- Puede que la latencia de una operación de acceso a archivos sea muy baja en las condiciones actuales, pero puede aumentar mucho en el futuro. Por ejemplo, se puede mover un archivo a un servidor que está a escala mundial.
- La sobrecarga resultante de usar la característica Async es pequeña.
- Las tareas asincrónicas se pueden ejecutar fácilmente en paralelo.

Ejecutar los ejemplos

Para ejecutar los ejemplos de este tema, puede crear una **aplicación WPF** o una **aplicación de Windows Forms** y luego agregar un **botón**. En el evento `Click` del botón, agregue una llamada al primer método de cada ejemplo.

En los ejemplos siguientes, se incluyen las directivas `using` siguientes.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Threading.Tasks;
```

Uso de la clase FileStream

En los ejemplos de este tema se usa la clase FileStream, que tiene una opción que hace que la E/S asincrónica se produzca en el nivel del sistema operativo. Si usa esta opción, puede evitar bloquear un subproceso ThreadPool en muchos casos. Para habilitar esta opción, especifique el argumento `useAsync=true` o `options=FileOptions.Asynchronous` en la llamada al constructor.

No puede usar esta opción con StreamReader y StreamWriter si los abre directamente al especificar una ruta de

acceso de archivo. En cambio, puede usar esta opción si les proporciona un Stream que ha abierto la clase FileStream. Tenga en cuenta que las llamadas asincrónicas son más rápidas en aplicaciones de interfaz de usuario aunque un subproceso ThreadPool se bloquee, porque el subproceso de interfaz de usuario no se bloquea durante la espera.

Escribir texto

En el ejemplo siguiente se escribe texto en un archivo. En cada instrucción await, el método finaliza inmediatamente. Cuando se complete la E/S de archivo, el método se reanuda en la instrucción que sigue a la instrucción await. Tenga en cuenta que el modificador async se encuentra en la definición de métodos que usan la instrucción await.

```
public async Task ProcessWriteAsync()
{
    string filePath = @"temp2.txt";
    string text = "Hello World\r\n";

    await WriteTextAsync(filePath, text);
}

private async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using (FileStream sourceStream = new FileStream(filePath,
        FileMode.Append, FileAccess.Write, FileShare.None,
        bufferSize: 4096, useAsync: true))
    {
        await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
    };
}
```

El ejemplo original incluye la instrucción `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, que es una contracción de las dos instrucciones siguientes:

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

La primera instrucción devuelve una tarea e inicia el procesamiento de archivos. La segunda instrucción con await finaliza el método inmediatamente y devuelve otra tarea. Después, cuando se complete el procesamiento de archivos, la ejecución vuelve a la instrucción que sigue a la instrucción await. Para obtener más información, vea Control Flow in Async Programs (C#) (Flujo de control en programas asincrónicos [C#]).

Leer texto

En el ejemplo siguiente se lee texto de un archivo. El texto se almacena en búfer y, en este caso, se coloca en un StringBuilder. A diferencia del ejemplo anterior, la evaluación de la instrucción await genera un valor. El método ReadAsync devuelve un Task<Int32>, por lo que la evaluación de la espera genera un valor `Int32` (`numRead`) después de que se complete la operación. Para obtener más información, consulte Tipos de valor devueltos asincrónicos (C#).

```
public async Task ProcessReadAsync()
{
    string filePath = @"temp2.txt";

    if (File.Exists(filePath) == false)
    {
        Debug.WriteLine("file not found: " + filePath);
    }
    else
    {
        try
        {
            string text = await ReadTextAsync(filePath);
            Debug.WriteLine(text);
        }
        catch (Exception ex)
        {
            Debug.WriteLine(ex.Message);
        }
    }
}

private async Task<string> ReadTextAsync(string filePath)
{
    using (FileStream sourceStream = new FileStream(filePath,
        FileMode.Open, FileAccess.Read, FileShare.Read,
        bufferSize: 4096, useAsync: true))
    {
        StringBuilder sb = new StringBuilder();

        byte[] buffer = new byte[0x1000];
        int numRead;
        while ((numRead = await sourceStream.ReadAsync(buffer, 0, buffer.Length)) != 0)
        {
            string text = Encoding.Unicode.GetString(buffer, 0, numRead);
            sb.Append(text);
        }

        return sb.ToString();
    }
}
```

E/S asincrónica en paralelo

En el ejemplo siguiente se muestra el procesamiento paralelo escribiendo 10 archivos de texto. Para cada archivo, el método WriteAsync devuelve una tarea que luego se agrega a una lista de tareas. La instrucción `await Task.WhenAll(tasks);` finaliza el método y se reanuda en el método cuando el procesamiento de archivos se completa para todas las tareas.

Tras completar las tareas, el ejemplo cierra todas las instancias de FileStream de un bloque `finally`. Si en lugar de ello, cada `FileStream` se ha creado en una instrucción `using`, la `FileStream` se podría desechar antes de completarse la tarea.

Tenga en cuenta que cualquier aumento del rendimiento se debe casi por completo al procesamiento en paralelo y no al procesamiento asincrónico. Las ventajas de la asincronía estriban en que no inmoviliza varios subprocesos ni el subproceso de interfaz de usuario.

```
public async Task ProcessWriteMultAsync()
{
    string folder = @"tempfolder\";
    List<Task> tasks = new List<Task>();
    List<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        for (int index = 1; index <= 10; index++)
        {
            string text = "In file " + index.ToString() + "\r\n";

            string fileName = "thefile" + index.ToString("00") + ".txt";
            string filePath = folder + fileName;

            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            FileStream sourceStream = new FileStream(filePath,
                FileMode.Append, FileAccess.Write, FileShare.None,
                bufferSize: 4096, useAsync: true);

            Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
            sourceStreams.Add(sourceStream);

            tasks.Add(theTask);
        }

        await Task.WhenAll(tasks);
    }

    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

Al usar los métodos WriteAsync y ReadAsync, puede especificar un CancellationToken, que puede usar para cancelar la operación en mitad de la secuencia. Para obtener más información, vea Ajustar una aplicación asincrónica (C#) y Cancelación en subprocesos administrados.

Vea también

- Programación asincrónica con Async y Await (C#)
- Tipos de valor devueltos asincrónicos (C#)
- Controlar el flujo en los programas asincrónicos (C#)

Atributos (C#)

29/06/2020 • 9 minutes to read • Edit Online

Los atributos proporcionan un método eficaz para asociar metadatos, o información declarativa, con código (ensamblados, tipos, métodos, propiedades, etc.). Después de asociar un atributo con una entidad de programa, se puede consultar el atributo en tiempo de ejecución mediante la utilización de una técnica denominada *reflexión*. Para obtener más información, vea Reflexión (C#).

Los atributos tienen las propiedades siguientes:

- Los atributos agregan metadatos al programa. Los *metadatos* son información sobre los tipos definidos en un programa. Todos los ensamblados .NET contienen un conjunto de metadatos específico que describe los tipos y miembros de tipo definidos en el ensamblado. Puede agregar atributos personalizados para especificar cualquier información adicional que sea necesaria. Para obtener más información, vea Crear atributos personalizados (C#).
- Puede aplicar uno o más atributos a todos los ensamblados, módulos o elementos de programa más pequeños como clases y propiedades.
- Los atributos pueden aceptar argumentos de la misma manera que los métodos y las propiedades.
- El programa puede examinar sus propios metadatos o los metadatos de otros programas mediante la reflexión. Para obtener más información, consulte Acceder a atributos mediante reflexión (C#).

Uso de atributos

Los atributos se pueden colocar en la mayoría de las declaraciones, aunque un determinado atributo podría restringir los tipos de declaraciones en que es válido. En C#, para especificar un atributo se coloca su nombre entre corchetes ([]) por encima de la declaración de la entidad a la que se aplica.

En este ejemplo, el atributo SerializableAttribute se usa para aplicar una característica específica a una clase:

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

Un método con el atributo DllImportAttribute se declara como en el siguiente ejemplo:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

En una declaración se puede colocar más de un atributo, como se muestra en el siguiente ejemplo:

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Algunos atributos se pueden especificar más de una vez para una entidad determinada. Un ejemplo de este tipo

de atributos multiuso es ConditionalAttribute:

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> **NOTE**
> Por convención, todos los nombres de atributos terminan con la palabra "Attribute" para distinguirlos de otros elementos de las bibliotecas de .NET. Sin embargo, no es necesario especificar el sufijo de atributo cuando utiliza atributos en el código. Por ejemplo, `[DllImport]` es equivalente a `[DllImportAttribute]`, pero `DllImportAttribute` es el nombre real del atributo en la biblioteca de clases .NET.

Parámetros de atributo

Muchos atributos tienen parámetros, que pueden ser posicionales, sin nombre o con nombre. Los parámetros posicionales deben especificarse en un orden determinado y no se pueden omitir. Los parámetros con nombre son opcionales y pueden especificarse en cualquier orden. Los parámetros posicionales se especifican en primer lugar. Por ejemplo, estos tres atributos son equivalentes:

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

El primer parámetro, el nombre del archivo DLL, es posicional y siempre va primero; los demás tienen un nombre. En este caso, ambos parámetros con nombre tienen el estado false de forma predeterminada, por lo que se pueden omitir. Los parámetros posicionales corresponden a los parámetros del constructor de atributos. Los parámetros con nombre u opcionales corresponden a propiedades o campos del atributo. Consulte la documentación del atributo individual para obtener información sobre los valores de parámetro predeterminados.

Destinos de atributo

El *destino* de un atributo es la entidad a la que se aplica dicho atributo. Por ejemplo, puede aplicar un atributo a una clase, un método determinado o un ensamblado completo. De forma predeterminada, el atributo se aplica al elemento que sigue. Pero puede identificar explícitamente, por ejemplo, si se aplica un atributo a un método, a su parámetro o a su valor devuelto.

Para identificar un destino de atributo de forma explícita, use la sintaxis siguiente:

```
[target : attribute-list]
```

La lista de posibles valores `target` se muestra en la tabla siguiente.

VALOR DEL OBJETIVO	SE APLICA A
`assembly`	Ensamblado completo
`module`	Módulo de ensamblado actual
`field`	Campo de una clase o un struct

VALOR DEL OBJETIVO	SE APLICA A
`event`	evento
`method`	Método o descriptores de acceso de propiedad `get` y `set`
`param`	Parámetros de método o parámetros de descriptor de acceso de propiedad `set`
`property`	Propiedad.
`return`	Valor devuelto de un método, indexador de propiedad o descriptor de acceso de propiedad `get`
`type`	Estructura, clase, interfaz, enumeración o delegado

Debe especificar el valor de destino `field` para aplicar un atributo al campo de respaldo creado para una propiedad implementada automáticamente.

En el ejemplo siguiente se muestra cómo aplicar atributos a ensamblados y módulos. Para obtener más información, vea Atributos comunes (C#).

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

En el ejemplo siguiente, se muestra cómo aplicar atributos a métodos, parámetros de método y valores devueltos por métodos en C#.

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to return value
[return: ValidatedContract]
int Method3() { return 0; }
```

NOTE

Independientemente de los destinos en los que `ValidatedContract` se define para que sea válido, debe especificarse el destino `return`, incluso si `ValidatedContract` se ha definido para que se aplique solo a los valores devueltos. En otras palabras, el compilador no usará información de `AttributeUsage` para resolver destinos de atributo ambiguos. Para obtener más información, consulte AttributeUsage (C#).

Usos comunes de los atributos

La lista siguiente incluye algunos de los usos comunes de atributos en el código:

- Marcar métodos con el atributo `WebMethod` en los servicios web para indicar que el método debe ser

invocable a través del protocolo SOAP. Para obtener más información, vea WebMethodAttribute.

- Describir cómo serializar parámetros de método al interoperar con código nativo. Para obtener más información, vea MarshalAsAttribute.
- Describir las propiedades COM para clases, métodos e interfaces.
- Llamar al código no administrado mediante la clase DllImportAttribute.
- Describir los ensamblados en cuanto a título, versión, descripción o marca.
- Describir qué miembros de una clase serializar para la persistencia.
- Describir cómo realizar asignaciones entre los miembros de clase y los nodos XML para la serialización XML.
- Describir los requisitos de seguridad para los métodos.
- Especificar las características utilizadas para reforzar la seguridad.
- Controlar optimizaciones mediante el compilador Just-In-Time (JIT) para que el código siga siendo fácil de depurar.
- Obtener información sobre el llamador de un método.

Secciones relacionadas

Para obtener más información, consulte:

- Crear atributos personalizados (C#)
- Acceder a atributos mediante reflexión (C#)
- Procedimiento para: Crear una unión de C/C++ mediante atributos (C#)
- Atributos comunes (C#)
- Información del llamador (C#)

Vea también

- Guía de programación de C#
- Reflexión (C#)
- Atributos
- Uso de atributos en C#

Crear atributos personalizados (C#)

29/06/2020 • 2 minutes to read • Edit Online

Para crear sus propios atributos personalizados, defina una clase de atributo derivada directa o indirectamente de Attribute, que agiliza y facilita la identificación de las definiciones de atributos en los metadatos. Imagínese que desea etiquetar tipos con el nombre del programador que los escribió. Puede definir una clase de atributos `Author` personalizada:

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class Author : System.Attribute
{
    private string name;
    public double version;

    public Author(string name)
    {
        this.name = name;
        version = 1.0;
    }
}
```

El nombre de la clase es el nombre del atributo, `Author`. Se deriva de `System.Attribute`, por lo que es una clase de atributo personalizada. Los parámetros del constructor son los parámetros posicionales del atributo personalizado. En este ejemplo, `name` es un parámetro posicional. Las propiedades o los campos públicos de lectura y escritura son parámetros con nombre. En este caso, `version` es el único parámetro con nombre. Observe el uso del atributo `AttributeUsage` para hacer que el atributo `Author` sea válido solo en las declaraciones de clase y de `struct`.

Puede usar este nuevo atributo de la siguiente manera:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` tiene un parámetro con nombre, `AllowMultiple`, con el que puede hacer que un atributo personalizado sea multiuso o de un solo uso. En el ejemplo de código siguiente se crea un atributo multiuso.

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // multiuse attribute
]
public class Author : System.Attribute
```

En el ejemplo de código siguiente se aplican varios atributos del mismo tipo a una clase.

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
    // P. Ackerman's code goes here...
    // R. Koch's code goes here...
}
```

Vea también

- System.Reflection
- Guía de programación de C#
- Escribir atributos personalizados
- Reflexión (C#)
- Atributos (C#)
- Acceder a atributos mediante reflexión (C#)
- AttributeUsage (C#)

Acceder a atributos mediante reflexión (C#)

18/03/2020 • 3 minutes to read • Edit Online

El hecho de que pueda definir atributos personalizados y colocarlos en el código fuente no serviría de mucho si no existiera ninguna forma de recuperar la información y actuar en consecuencia. Mediante la reflexión, puede recuperar la información que se ha definido con atributos personalizados. El método clave es `GetCustomAttributes`, que devuelve una matriz de objetos que son los equivalentes en tiempo de ejecución de los atributos de código fuente. Este método tiene varias versiones sobrecargadas. Para obtener más información, consulta Attribute.

Una especificación de atributo como:

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
```

es conceptualmente equivalente a esta:

```
Author anonymousAuthorObject = new Author("P. Ackerman");
anonymousAuthorObject.version = 1.1;
```

En cambio, el código no se ejecuta hasta que se consulta a `SampleClass` sobre los atributos. Llamar a `GetCustomAttributes` en `SampleClass` hace que se cree e inicialice un objeto `Author` como se ha mostrado anteriormente. Si la clase tiene otros atributos, se crean otros objetos de atributo de forma similar. Luego, `GetCustomAttributes` devuelve el objeto `Author` y cualquier otro objeto de atributo en una matriz. Después, puede recorrer en iteración esta matriz, determinar qué atributos se han aplicado según el tipo de cada elemento de la matriz y extraer información de los objetos de atributo.

Ejemplo

Este es un ejemplo completo. Se define un atributo personalizado, se aplica a varias entidades y se recupera mediante reflexión.

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class Author : System.Attribute
{
    string name;
    public double version;

    public Author(string name)
    {
        this.name = name;

        // Default value.
        version = 1.0;
    }

    public string GetName()
    {
        return name;
    }
}
```

```
// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine("Author information for {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);  // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("   {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}
/* Output:
    Author information for FirstClass
       P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
       R. Koch, version 2.00
       P. Ackerman, version 1.00
*/
```

Vea también

- System.Reflection
- Attribute
- Guía de programación de C#
- Retrieving Information Stored in Attributes (Recuperar la información almacenada en atributos)
- Reflexión (C#)

- Atributos (C#)
- Crear atributos personalizados (C#)

Procedimiento para crear una unión de C o C++ mediante atributos (C#)

18/03/2020 • 2 minutes to read • Edit Online

Mediante el uso de atributos, puede personalizar la manera en que los structs se disponen en la memoria. Por ejemplo, puede crear lo que se conoce como una unión en C/ C++ mediante los atributos `StructLayout(LayoutKind.Explicit)` y `FieldOffset`.

Ejemplo

En este segmento de código, todos los campos de `TestUnion` empiezan en la misma ubicación en la memoria.

```
// Add a using directive for System.Runtime.InteropServices.

[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

Ejemplo

A continuación se muestra otro ejemplo en el que los campos empiezan en ubicaciones diferentes establecidas explícitamente.

```
// Add a using directive for System.Runtime.InteropServices.

[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
    public byte b;
}
```

Los dos campos enteros, `i1` e `i2`, tiene las mismas ubicaciones en la memoria que `lg`. Este tipo de control sobre el diseño del struct es útil cuando se usa la invocación de plataforma.

Vea también

- System.Reflection
- Attribute
- Guía de programación de C#
- Atributos
- Reflexión (C#)
- Atributos (C#)
- Crear atributos personalizados (C#)
- Acceder a atributos mediante reflexión (C#)

Colecciones (C#)

29/06/2020 • 22 minutes to read • Edit Online

Para muchas aplicaciones, puede que desee crear y administrar grupos de objetos relacionados. Existen dos formas de agrupar objetos: mediante la creación de matrices de objetos y con la creación de colecciones de objetos.

Las matrices son muy útiles para crear y trabajar con un número fijo de objetos fuertemente tipados. Para obtener información sobre las matrices, vea Matrices.

Las colecciones proporcionan una manera más flexible de trabajar con grupos de objetos. A diferencia de las matrices, el grupo de objetos con el que trabaja puede aumentar y reducirse de manera dinámica a medida que cambian las necesidades de la aplicación. Para algunas colecciones, puede asignar una clave a cualquier objeto que incluya en la colección para, de este modo, recuperar rápidamente el objeto con la clave.

Una colección es una clase, por lo que debe declarar una instancia de la clase para poder agregar elementos a dicha colección.

Si la colección contiene elementos de un solo tipo de datos, puede usar una de las clases del espacio de nombres System.Collections.Generic. Una colección genérica cumple la seguridad de tipos para que ningún otro tipo de datos se pueda agregar a ella. Cuando recupera un elemento de una colección genérica, no tiene que determinar su tipo de datos ni convertirlo.

> **NOTE**
> Para los ejemplos de este tema, incluya las directivas using para los espacios de nombres `System.Collections.Generic` y `System.Linq`.

En este tema

- Uso de una colección Simple
- Tipos de colecciones
  - Clases System.Collections.Generic
  - Clases System.Collections.Concurrent
  - Clases System.Collections
- Implementación de una colección de pares de clave/valor
- Uso de LINQ para tener acceso a una colección
- Ordenar una colección
- Definición de una colección personalizada
- Iteradores

Uso de una colección Simple

Los ejemplos de esta sección usan la clase genérica List<T>, que le permite trabajar con una lista de objetos fuertemente tipados.

En el ejemplo siguiente se crea una lista de cadenas y luego se recorren en iteración mediante una instrucción foreach.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Si el contenido de una colección se conoce de antemano, puede usar un *inicializador de colección* para inicializar la colección. Para obtener más información, vea Inicializadores de objeto y colección.

El ejemplo siguiente es el mismo que el ejemplo anterior, excepto que se usa un inicializador de colección para agregar elementos a la colección.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Puede usar una instrucción for en lugar de una instrucción `foreach` para recorrer en iteración una colección. Esto se consigue con el acceso a los elementos de la colección mediante la posición de índice. El índice de los elementos comienza en 0 y termina en el número de elementos menos 1.

El ejemplo siguiente recorre en iteración los elementos de una colección mediante `for` en lugar de `foreach`.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

El ejemplo siguiente quita un elemento de la colección especificando el objeto que se quitará.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

El ejemplo siguiente quita elementos de una lista genérica. En lugar de una instrucción `foreach`, se usa una instrucción for que procesa una iteración en orden descendente. Esto es porque el método RemoveAt hace que los elementos después de un elemento quitado tengan un valor de índice inferior.

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

Para el tipo de elementos de List<T>, también puede definir su propia clase. En el ejemplo siguiente, la clase `Galaxy` que usa List<T> se define en el código.

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
        {
            new Galaxy() { Name="Tadpole", MegaLightYears=400},
            new Galaxy() { Name="Pinwheel", MegaLightYears=25},
            new Galaxy() { Name="Milky Way", MegaLightYears=0},
            new Galaxy() { Name="Andromeda", MegaLightYears=3}
        };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

Tipos de colecciones

.NET proporciona muchas colecciones comunes. Cada tipo de colección está diseñado para un fin específico.

En esta sección se describen algunas de las clases de colecciones comunes:

- Clases System.Collections.Generic
- Clases System.Collections.Concurrent
- Clases System.Collections

Clases System.Collections.Generic

Puede crear una colección genérica mediante una de las clases del espacio de nombres System.Collections.Generic. Una colección genérica es útil cuando todos los elementos de la colección tienen el mismo tipo. Una colección genérica exige el establecimiento de fuertes tipos al permitir agregar solo los tipos de datos deseados.

En la tabla siguiente se enumeran algunas de las clases usadas con frecuencia del espacio de nombres System.Collections.Generic:

CLASE	DESCRIPCIÓN
Dictionary<TKey,TValue>	Representa una colección de pares de clave y valor que se organizan según la clave.
List<T>	Representa una lista de objetos a los que puede tener acceso el índice. Proporciona métodos para buscar, ordenar y modificar listas.
Queue<T>	Representa una colección de objetos de primeras entradas, primeras salidas (FIFO).

CLASE	DESCRIPCIÓN
SortedList<TKey,TValue>	Representa una colección de pares clave-valor que se ordenan por claves según la implementación de IComparer<T> asociada.
Stack<T>	Representa una colección de objetos de últimas entradas, primeras salidas (LIFO).

Para obtener más información, vea Tipos de colección utilizados normalmente, Seleccionar una clase de colección y System.Collections.Generic.

Clases System.Collections.Concurrent

En .NET Framework 4 y versiones posteriores, las colecciones del espacio de nombres System.Collections.Concurrent proporcionan operaciones eficaces y seguras para subprocesos con el fin de acceder a los elementos de la colección desde varios subprocesos.

Las clases del espacio de nombres System.Collections.Concurrent deben usarse en lugar de los tipos correspondientes de los espacios de nombres System.Collections.Generic y System.Collections cada vez que varios subprocesos tengan acceso de manera simultánea a la colección. Para obtener más información, vea Colecciones seguras para subprocesos y System.Collections.Concurrent.

Algunas clases incluidas en el espacio de nombres System.Collections.Concurrent son BlockingCollection<T>, ConcurrentDictionary<TKey,TValue>, ConcurrentQueue<T> y ConcurrentStack<T>.

Clases System.Collections

Las clases del espacio de nombres System.Collections no almacenan los elementos como objetos de tipo específico, sino como objetos del tipo `Object`.

Siempre que sea posible, debe usar las colecciones genéricas del espacio de nombres System.Collections.Generic o del espacio de nombres System.Collections.Concurrent en lugar de los tipos heredados del espacio de nombres `System.Collections`.

En la siguiente tabla se enumeran algunas de las clases usadas con frecuencia en el espacio de nombres `System.Collections`:

CLASE	DESCRIPCIÓN
ArrayList	Representa una matriz cuyo tamaño aumenta dinámicamente cuando es necesario.
Hashtable	Representa una colección de pares de clave y valor que se organizan por código hash de la clave.
Queue	Representa una colección de objetos de primeras entradas, primeras salidas (FIFO).
Stack	Representa una colección de objetos de últimas entradas, primeras salidas (LIFO).

El espacio de nombres System.Collections.Specialized proporciona clases de colección especializadas y fuertemente tipadas como, por ejemplo, colecciones de solo cadena y diccionarios híbridos y de lista vinculada.

Implementación de una colección de pares de clave/valor

La colección genérica Dictionary<TKey,TValue> le permite tener acceso a los elementos de una colección con la

clave de cada elemento. Cada adición al diccionario consta de un valor y de su clave asociada. Recuperar un valor usando su clave es rápido porque la clase `Dictionary` se implementa como una tabla hash.

En el ejemplo siguiente se crea una colección `Dictionary` y se recorre en iteración el diccionario usando una instrucción `foreach`.

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Para usar un inicializador de colección para compilar la colección `Dictionary`, puede reemplazar los métodos `BuildDictionary` y `AddToDictionary` por el método siguiente.

```
private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}
```

En el ejemplo siguiente se usa el método ContainsKey y la propiedad Item[] de `Dictionary` para encontrar rápidamente un elemento por clave. La propiedad `Item` le permite tener acceso a un elemento de la colección `elements` usando `elements[symbol]` en C#.

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

En el ejemplo siguiente se usa en su lugar el método TryGetValue para encontrar rápidamente un elemento por clave.

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

Uso de LINQ para tener acceso a una colección

LINQ (Language-Integrated Query) puede usar para tener acceso a las colecciones. Las consultas LINQ proporcionan capacidades de filtrado, ordenación y agrupación. Para obtener más información, vea Introducción a LINQ en C#.

El ejemplo siguiente ejecuta una consulta LINQ en una `List` genérica. La consulta LINQ devuelve otra colección que contiene los resultados.

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Ordenar una colección

En el ejemplo siguiente se muestra un procedimiento para ordenar una colección. El ejemplo ordena las instancias de la clase `Car` que se almacenan en un List<T>. La clase `Car` implementa la interfaz IComparable<T>, que requiere implementar el método CompareTo.

Cada llamada al método CompareTo realiza una comparación única que se usa para la ordenación. El código escrito por el usuario en el método `CompareTo` devuelve un valor para cada comparación del objeto actual con otro objeto. El valor devuelto es menor que cero si el objeto actual es menor que el otro objeto, mayor que cero si el objeto actual es mayor que el otro objeto y cero si son iguales. Esto permite definir en el código los criterios de mayor que, menor que e igual.

En el método `ListCars`, la instrucción `cars.Sort()` ordena la lista. Esta llamada al método Sort de List<T> hace que se llame automáticamente al método `CompareTo` para los objetos `Car` de `List`.

```
private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
```

```
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.Write(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    //  blue  50 car4
    //  blue  30 car5
    //  blue  20 car1
    //  green 50 car7
    //  green 10 car3
    //  red   60 car6
    //  red   50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}
```

Definición de una colección personalizada

Puede definir una colección implementando la interfaz IEnumerable<T> o IEnumerable.

Aunque puede definir una colección personalizada, es mejor usar las colecciones incluidas en .NET. Estas colecciones se describen en la sección Tipos de colecciones de este artículo.

En el siguiente ejemplo se define una clase de colección personalizada denominada `AllColors`. Esta clase implementa la interfaz IEnumerable que requiere implementar el método GetEnumerator.

El método `GetEnumerator` devuelve una instancia de la clase `ColorEnumerator`. `ColorEnumerator` implementa la interfaz IEnumerator, que requiere que la propiedad Current, el método MoveNext y el método Reset estén implementados.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
```

```
    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}
```

Iterators

Los *iteradores* se usan para efectuar una iteración personalizada en una colección. Un iterador puede ser un método o un descriptor de acceso `get`. Un iterador usa una instrucción yield return para devolver cada elemento de la colección a la vez.

Llame a un iterador mediante una instrucción foreach. Cada iteración del bucle `foreach` llama al iterador. Cuando se alcanza una instrucción `yield return` en el iterador, se devuelve una expresión y se conserva la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama al iterador.

Para obtener más información, vea Iteradores (C#).

El siguiente ejemplo usa el método del iterador. El método del iterador tiene una instrucción `yield return` que se encuentra dentro de un bucle for. En el método `ListEvenNumbers`, cada iteración del cuerpo de la instrucción `foreach` crea una llamada al método iterador, que continúa con la siguiente instrucción `yield return`.

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Vea también

- Inicializadores de objeto y colección
- Conceptos de programación (C#)
- Option Strict (instrucción)
- LINQ to Objects (C#)
- Parallel LINQ (PLINQ)
- Colecciones y estructuras de datos
- Seleccionar una clase de colección
- Comparaciones y ordenaciones en colecciones

- Cuándo utilizar colecciones genéricas

Covarianza y contravarianza (C#)

29/06/2020 • 5 minutes to read • Edit Online

En C#, la covarianza y la contravarianza habilitan la conversión de referencias implícita de tipos de matriz, tipos de delegado y argumentos de tipo genérico. La covarianza conserva la compatibilidad de asignaciones y la contravarianza la invierte.

El siguiente código muestra la diferencia entre la compatibilidad de asignaciones, la covarianza y la contravarianza.

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

La covarianza de matrices permite la conversión implícita de una matriz de un tipo más derivado a una matriz de un tipo menos derivado. Pero esta operación no es segura, tal como se muestra en el ejemplo de código siguiente.

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

La compatibilidad de la covarianza y la contravarianza con grupos de métodos permite hacer coincidir firmas de método con tipos de delegado. Esto le permite asignar a los delegados no solo métodos con firmas coincidentes, sino métodos que devuelven tipos más derivados (covarianza) o que aceptan parámetros con tipos menos derivados (contravarianza) que el especificado por el tipo de delegado. Para obtener más información, vea Varianza en delegados (C#) y Usar varianza en delegados (C#).

En el ejemplo de código siguiente, se muestra la compatibilidad de covarianza y contravarianza con grupos de métodos.

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

En .NET Framework 4 o versiones posteriores, C# admite la covarianza y contravarianza en las interfaces genéricas y los delegados, y permite la conversión implícita de los parámetros de tipo genérico. Para obtener más información, vea Varianza en interfaces genéricas (C#) y Varianza en delegados (C#).

En el ejemplo de código siguiente, se muestra la conversión implícita de referencias para interfaces genéricas.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

Un delegado o interfaz genéricos se denominan *variante* si sus parámetros genéricos se declaran como covariantes o contravariantes. C# le permite crear sus propias interfaces y delegados variantes. Para obtener más información, consulte Crear interfaces genéricas variantes (C#) y Varianza en delegados (C#).

Temas relacionados

TITLE	DESCRIPCIÓN
Varianza en interfaces genéricas (C#)	Describe la covarianza y contravarianza en las interfaces genéricas y proporciona una lista de interfaces genéricas variantes en .NET Framework.
Crear interfaces genéricas variantes (C#)	Se muestra cómo crear interfaces variantes personalizadas.
Usar la varianza en interfaces para las colecciones genéricas (C#)	Se muestra cómo la compatibilidad de covarianza y contravarianza en las interfaces IEnumerable<T> y IComparable<T> puede ayudarle a volver a usar el código.
Varianza en delegados (C#)	Se describe la covarianza y contravarianza en delegados genéricos y no genéricos y se proporciona una lista de delegados genéricos variantes en .NET Framework.
Usar varianza en delegados (C#)	Se muestra cómo usar la compatibilidad de covarianza y contravarianza en los delegados no genéricos para que coincidan las firmas de método con los tipos de delegado.
Usar varianza para los delegados genéricos Func y Action (C#)	Se muestra cómo la compatibilidad de covarianza y contravarianza en los delegados `Func` y `Action` puede ayudarle a volver a usar el código.

Varianza en interfaces genéricas (C#)

29/06/2020 • 4 minutes to read • Edit Online

En .NET Framework 4 se ha presentado la compatibilidad con la varianza para varias interfaces genéricas existentes. La compatibilidad con la varianza permite la conversión implícita de clases que implementan estas interfaces.

A partir de .NET Framework 4, las siguientes interfaces son variantes:

- IEnumerable<T> (T es covariante)
- IEnumerator<T> (T es covariante)
- IQueryable<T> (T es covariante)
- IGrouping<TKey,TElement> (`TKey` y `TElement` son covariantes)
- IComparer<T> (T es contravariante)
- IEqualityComparer<T> (T es contravariante)
- IComparable<T> (T es contravariante)

A partir de .NET Framework 4.5, las siguientes interfaces son variantes:

- IReadOnlyList<T> (T es covariante)
- IReadOnlyCollection<T> (T es covariante)

La covarianza permite que un método tenga un tipo de valor devuelto más derivado que los que se definen en los parámetros de tipo genérico de la interfaz. Para ilustrar la característica de la covarianza, considere estas interfaces genéricas: `IEnumerable<Object>` y `IEnumerable<String>`. La interfaz `IEnumerable<String>` no hereda la interfaz `IEnumerable<Object>`. En cambio, el tipo `String` hereda el tipo `Object`, y en algunos casos puede que quiera asignar objetos de estas interfaces entre sí. Esto se muestra en el ejemplo de código siguiente.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

En versiones anteriores de .NET Framework, este código provoca un error de compilación en C# y, si `Option Strict` está activado, en Visual Basic. Pero ahora puede usar `strings` en lugar de `objects`, como se muestra en el ejemplo anterior, porque la interfaz IEnumerable<T> es covariante.

La contravarianza permite que un método tenga tipos de argumento menos derivados que los que se especifican en el parámetro genérico de la interfaz. Para ilustrar la contravarianza, se presupone que ha creado una clase `BaseComparer` para comparar instancias de la clase `BaseClass`. La clase `BaseComparer` implementa la interfaz `IEqualityComparer<BaseClass>`. Como la interfaz IEqualityComparer<T> ahora es contravariante, puede usar `BaseComparer` para comparar instancias de clases que heredan la clase `BaseClass`. Esto se muestra en el ejemplo de código siguiente.

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

Para obtener más ejemplos, vea Usar la varianza en interfaces para las colecciones genéricas (C#).

La varianza para interfaces genéricas solo es compatible con tipos de referencia. Los tipos de valor no admiten la varianza. Por ejemplo, `IEnumerable<int>` no puede convertirse implícitamente en `IEnumerable<object>`, porque los enteros se representan mediante un tipo de valor.

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

También es importante recordar que las clases que implementan las interfaces variantes siguen siendo invariables. Por ejemplo, aunque List<T> implementa la interfaz covariante IEnumerable<T>, no puede convertir `List<String>` en `List<Object>` implícitamente. Esto se muestra en el siguiente código de ejemplo.

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

Vea también

- Usar la varianza en interfaces para las colecciones genéricas (C#)
- Crear interfaces genéricas variantes (C#)
- Interfaces genéricas
- Varianza en delegados (C#)

Crear interfaces genéricas variantes (C#)

29/06/2020 • 8 minutes to read • Edit Online

Puede declarar parámetros de tipo genérico en las interfaces como covariantes o contravariantes. La *covarianza* permite que los métodos de interfaz tengan tipos de valor devuelto más derivados que los que se definen en los parámetros de tipo genérico. La *contravarianza* permite que los métodos de interfaz tengan tipos de argumento menos derivados que los que se especifican en los parámetros genéricos. Las interfaces genéricas que tienen parámetros de tipo genérico covariantes o contravariantes se llaman *variantes*.

> **NOTE**
>
> En .NET Framework 4 se ha presentado la compatibilidad con la varianza para varias interfaces genéricas existentes. Para ver la lista de interfaces variantes de .NET, vea Varianza en interfaces genéricas (C#).

Declarar interfaces genéricas variantes

Puede declarar interfaces genéricas variantes mediante las palabras clave `in` y `out` para los parámetros de tipo genérico.

> **IMPORTANT**
>
> Los parámetros `ref`, `in` y `out` de C# no pueden ser variantes. Los tipos de valor tampoco admiten la varianza.

Puede declarar un parámetro de tipo genérico covariante mediante la palabra clave `out`. El tipo covariante debe cumplir las siguientes condiciones:

- El tipo se usa únicamente como tipo de valor devuelto de los métodos de interfaz, y no como tipo de los argumentos de método. Esto se muestra en el siguiente ejemplo, en el que el tipo `R` se declara como covariante.

  ```
  interface ICovariant<out R>
  {
      R GetSomething();
      // The following statement generates a compiler error.
      // void SetSomething(R sampleArg);

  }
  ```

  Hay una excepción para esta regla. Si tiene un delegado genérico contravariante como parámetro de método, puede usar el tipo como parámetro de tipo genérico para el delegado. Esto se muestra en el siguiente ejemplo con el tipo `R`. Para obtener más información, vea Varianza en delegados (C#) y Usar la varianza para los delegados genéricos Func y Action (C#).

  ```
  interface ICovariant<out R>
  {
      void DoSomething(Action<R> callback);
  }
  ```

- El tipo no se usa como restricción genérica para los métodos de interfaz. Esto se muestra en el siguiente código.

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

Puede declarar un parámetro de tipo genérico contravariante mediante la palabra clave `in`. El tipo contravariante solo se puede usar como tipo de los argumentos de método, y no como tipo de valor devuelto de los métodos de interfaz. El tipo contravariante también se puede usar para las restricciones genéricas. En el siguiente código se muestra cómo declarar una interfaz contravariante y cómo usar una restricción genérica para uno de sus métodos.

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

También se puede admitir la covarianza y la contravarianza en la misma interfaz, pero para distintos parámetros de tipo, como se muestra en el siguiente ejemplo de código.

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

Implementar interfaces genéricas variantes

Las interfaces genéricas variantes se implementan en las clases usando la misma sintaxis que se usa para las interfaces invariables. En el siguiente ejemplo de código se muestra cómo implementar una interfaz covariante en una clase genérica.

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Las clases que implementan interfaces variantes son invariables. Por ejemplo, considere el fragmento de código siguiente:

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Extender interfaces genéricas variantes

Al extender una interfaz genérica variante, debe usar las palabras clave `in` y `out` para especificar de forma explícita si la interfaz derivada admite la varianza. El compilador no infiere la varianza de la interfaz que se va a extender. Por ejemplo, observe las siguientes interfaces.

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

En la interfaz `IInvariant<T>`, el parámetro de tipo genérico `T` es invariable, mientras que en `IExtCovariant<out T>` el parámetro de tipo es covariante, si bien ambas interfaces extienden la misma interfaz. La misma regla se aplica a los parámetros de tipo genérico contravariantes.

Puede crear una interfaz que extienda la interfaz donde el parámetro de tipo genérico `T` es covariante y la interfaz donde es contravariante si, en la interfaz que va a extender, el parámetro de tipo genérico `T` es invariable. Esto se muestra en el siguiente código de ejemplo.

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

Pero si un parámetro de tipo genérico `T` se declara como covariante en una interfaz, no puede declararlo como contravariante en la interfaz extensible (o viceversa). Esto se muestra en el siguiente código de ejemplo.

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Evitar la ambigüedad

Al implementar interfaces genéricas variantes, la varianza a veces puede implicar ambigüedad. Debe evitarse esta ambigüedad.

Por ejemplo, si implementa explícitamente en una clase la misma interfaz genérica variante con distintos parámetros de tipo genérico, puede crear ambigüedad. El compilador no genera ningún error en este caso, pero no se especifica qué implementación de interfaz se va a elegir en tiempo de ejecución. Esta ambigüedad podría provocar errores sutiles en el código. Observe el siguiente ejemplo de código.

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

En este ejemplo no se especifica cómo elige el método `pets.GetEnumerator` entre `Cat` y `Dog`. Esto podría producir problemas en el código.

Vea también

- Varianza en interfaces genéricas (C#)
- Usar varianza para los delegados genéricos Func y Action (C#)

Usar la varianza en interfaces para las colecciones genéricas (C#)

29/06/2020 • 3 minutes to read • Edit Online

Una interfaz covariante permite que sus métodos devuelvan tipos más derivados que los especificados en la interfaz. Una interfaz contravariante permite que sus métodos acepten parámetros de tipos menos derivados que los especificados en la interfaz.

Varias interfaces existentes en .NET Framework 4 pasaron a ser covariantes y contravariantes. Por ejemplo, IEnumerable<T> y IComparable<T>. Esto permite volver a usar métodos que funcionan con colecciones genéricas de tipos base para colecciones de tipos derivados.

Para ver una lista de interfaces variantes de .NET, vea Varianza en interfaces genéricas (C#).

Convertir colecciones genéricas

En el ejemplo siguiente se muestran las ventajas de la compatibilidad con la covarianza en la interfaz IEnumerable<T>. El método `PrintFullName` acepta una colección del tipo `IEnumerable<Person>` como parámetro. Pero se puede volver a usar para una colección del tipo `IEnumerable<Employee>` porque `Employee` hereda `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
            person.FirstName, person.LastName);
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}
```

Comparar colecciones genéricas

En el ejemplo siguiente se muestran las ventajas de la compatibilidad con la contravarianza en la interfaz IComparer<T>. La clase `PersonComparer` implementa la interfaz `IComparer<Person>`. Pero se puede volver a usar esta clase para comparar una secuencia de objetos del tipo `Employee` porque `Employee` hereda `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName = "Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

Vea también

- Varianza en interfaces genéricas (C#)

Varianza en delegados (C#)

29/06/2020 • 9 minutes to read • Edit Online

En .NET Framework 3.5 se presentó por primera vez la compatibilidad con la varianza para hacer coincidir firmas de método con tipos de delegados en todos los delegados en C#. Esto significa que puede asignar a los delegados no solo métodos con firmas coincidentes, sino métodos que devuelven tipos más derivados (covarianza) o que aceptan parámetros con tipos menos derivados (contravarianza) que el especificado por el tipo de delegado. Esto incluye delegados genéricos y no genéricos.

Por ejemplo, consideremos el siguiente código, que tiene dos clases y dos delegados: genéricos y no genéricos.

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

Al crear delegados de los tipos `SampleDelegate` o `SampleGenericDelegate<A, R>`, puede asignar uno de los métodos siguientes a dichos delegados.

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

En el ejemplo de código siguiente se ilustra la conversión implícita entre la firma del método y el tipo de delegado.

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

Para obtener más ejemplos, vea Usar varianza en delegados (C#) y Usar la varianza para los delegados genéricos Func y Action (C#).

Varianza en parámetros de tipo genérico

En .NET Framework 4 o posterior puede habilitar la conversión implícita entre los delegados, de modo que los delegados genéricos con tipos diferentes especificados por parámetros de tipo genérico se puedan asignar entre sí, en el caso de que los tipos se hereden entre sí, como requiere la varianza.

Para habilitar la conversión implícita, debe declarar explícitamente parámetros genéricos en un delegado como covariante o contravariante mediante la palabra clave `in` o `out`.

En el ejemplo de código siguiente se muestra cómo se crea un delegado que tiene un parámetro de tipo genérico covariante.

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

Si usa solo la compatibilidad con la varianza para hacer coincidir firmas de método con tipos de delegados y no usa las palabras clave `in` y `out`, es posible que en algunas ocasiones pueda crear instancias de delegados con métodos o expresiones lambda idénticos, pero no pueda asignar un delegado a otro.

En el ejemplo de código siguiente, `SampleGenericDelegate<String>` no se puede convertir explícitamente a `SampleGenericDelegate<Object>`, aunque `String` hereda `Object`. Para solucionar este problema, marque el parámetro genérico `T` con la palabra clave `out`.

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Delegados genéricos con parámetros de tipo variante en .NET

En .NET Framework 4 se presentó por primera vez la compatibilidad con la varianza para los parámetros de tipo genérico en varios delegados genéricos existentes:

- `Action` delega del espacio de nombres System, por ejemplo, Action<T> y Action<T1,T2>

- `Func` delega del espacio de nombres System, por ejemplo, Func<TResult> y Func<T,TResult>
- Delegado Predicate<T>.
- Delegado Comparison<T>.
- Delegado Converter<TInput,TOutput>.

Para obtener más información y ejemplos, vea Using Variance for Func and Action Generic Delegates (C#) (Usar varianza para delegados genéricos Func y Action (C#)).

Declarar parámetros de tipo variante en delegados genéricos

Si un delegado genérico tiene parámetros de tipo genérico covariante o contravariante, se puede hacer referencia a él como un *delegado genérico variante*.

Puede declarar un parámetro de tipo genérico covariante en un delegado genérico mediante la palabra clave `out`. El tipo covariante puede usarse solo como un tipo de valor devuelto de método, y no como un tipo de argumentos de método. En el siguiente ejemplo de código se muestra cómo declarar un delegado genérico covariante.

```
public delegate R DCovariant<out R>();
```

Puede declarar un parámetro de tipo genérico contravariante en un delegado genérico mediante la palabra clave `in`. El tipo contravariante puede usarse solo como un tipo de argumentos de método, y no como un tipo de valor devuelto de método. En el siguiente ejemplo de código se muestra cómo declarar un delegado genérico contravariante.

```
public delegate void DContravariant<in A>(A a);
```

> **IMPORTANT**
> Los parámetros `ref`, `in` y `out` de C# no se pueden marcar como variantes.

También es posible admitir la varianza y la covarianza en el mismo delegado, pero para parámetros de tipo diferentes. Esta implementación se muestra en el ejemplo siguiente.

```
public delegate R DVariant<in A, out R>(A a);
```

Crear instancias de delegados genéricos variantes e invocarlos

Puede crear instancias de delegados variantes e invocarlos del mismo modo que crea instancias de delegados invariables y los invoca. En el ejemplo siguiente, se crea una instancia del delegado mediante una expresión lambda.

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Combinar delegados genéricos variantes

No combine delegados variantes. El método Combine no admite la conversión de delegados variantes y espera que los delegados sean exactamente del mismo tipo. Esto puede provocar una excepción en tiempo de ejecución cuando se combinan delegados mediante el método Combine o mediante el operador `+`, como se muestra en el ejemplo de código siguiente.

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Varianza en parámetros de tipo genérico para los tipos de referencia y valor

La varianza para parámetros de tipo genérico solo es compatible con tipos de referencia. Por ejemplo, `DVariant<int>` no se puede convertir implícitamente en `DVariant<Object>` o `DVariant<long>`, porque un entero es un tipo de valor.

En el ejemplo siguiente se muestra que la varianza en parámetros de tipo genérico no se admite para tipos de valor.

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

Vea también

- Genéricos
- Usar varianza para los delegados genéricos Func y Action (C#)
- Procedimiento para combinar delegados (delegados de multidifusión)

Usar varianza en delegados (C#)

18/03/2020 • 3 minutes to read • Edit Online

Al asignar un método a un delegado, la *covarianza* y la *contravarianza* proporcionan flexibilidad para hacer coincidir un tipo de delegado con una firma de método. La covarianza permite que un método tenga un tipo de valor devuelto más derivado que el definido en el delegado. La contravarianza permite un método que tiene tipos de parámetro menos derivados que los del tipo de delegado.

Ejemplo 1: covarianza

Description

En este ejemplo se muestra cómo se pueden usar delegados con métodos que tienen tipos de valor devuelto derivados del tipo de valor devuelto en la firma del delegado. El tipo de datos devuelto por `DogsHandler` es de tipo `Dogs`, que se deriva del tipo `Mammals` definido en el delegado.

Código

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

Ejemplo 2: contravarianza

Description

En este ejemplo se muestra cómo se pueden usar delegados con métodos que tienen parámetros que son tipos base del tipo de parámetro de la firma del delegado. Con la contravarianza, puede usar un controlador de eventos en lugar de controladores independientes. En el ejemplo siguiente se usan dos delegados:

- Un delegado KeyEventHandler que define la firma del evento Button.KeyDown. Su firma es:

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- Un delegado MouseEventHandler que define la firma del evento Button.MouseClick. Su firma es:

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e)
```

En el ejemplo se define un controlador de eventos con un parámetro EventArgs, que se usa para controlar los eventos `Button.KeyDown` y `Button.MouseClick`. Es posible hacer esto porque EventArgs es un tipo base de KeyEventArgs y MouseEventArgs.

Código

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;

}
```

Vea también

- Varianza en delegados (C#)
- Usar varianza para los delegados genéricos Func y Action (C#)

Usar la varianza para los delegados genéricos Func y Action (C#)

18/03/2020 • 2 minutes to read • Edit Online

En estos ejemplos se muestra cómo usar la covarianza y la contravarianza en los delegados genéricos `Func` y `Action` para habilitar la reutilización de métodos y proporcionar más flexibilidad en el código.

Para obtener más información sobre la covarianza y la contravarianza, vea Varianza en delegados (C#)

Usar delegados con parámetros de tipo covariante

En el ejemplo siguiente se muestran las ventajas de la compatibilidad con la covarianza en los delegados `Func` genéricos. El método `FindByTitle` toma un parámetro del tipo `String` y devuelve un objeto del tipo `Employee`. Pero este método se puede asignar al delegado `Func<String, Person>` porque `Employee` hereda `Person`.

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Usar delegados con parámetros de tipo contravariante

En el ejemplo siguiente se muestran las ventajas de la compatibilidad con la contravarianza en los delegados `Action` genéricos. El método `AddToContacts` toma un parámetro del tipo `Person`. Pero este método se puede asignar al delegado `Action<Employee>` porque `Employee` hereda `Person`.

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

Vea también

- Covarianza y contravarianza (C#)
- Genéricos

Árboles de expresión (C#)

29/06/2020 • 8 minutes to read • Edit Online

Los árboles de expresión representan el código en una estructura de datos en forma de árbol donde cada nodo es una expresión, por ejemplo, una llamada a método o una operación binaria como `x < y`.

El código representado en árboles de expresión se puede compilar y ejecutar. Esto permite realizar cambios dinámicos en el código ejecutable, ejecutar consultas LINQ en varias bases de datos y crear consultas dinámicas. Para obtener más información sobre los árboles de expresión en LINQ, consulte Procedimiento para usar árboles de expresión para crear consultas dinámicas (C#).

Los árboles de expresión también se usan en Dynamic Language Runtime (DLR) para proporcionar interoperabilidad entre los lenguajes dinámicos y .NET y, asimismo, para permitir que los programadores de compiladores emitan árboles de expresión en lugar de Lenguaje intermedio de Microsoft (MSIL). Para obtener más información sobre el entorno DLR, vea Información general acerca de Dynamic Language Runtime.

Puede hacer que el compilador de Visual Basic o de C# cree un árbol de expresión en función de una expresión lambda anónima, o bien puede crear árboles de expresión manualmente usando el espacio de nombres System.Linq.Expressions.

Crear árboles de expresión a partir de expresiones lambda

Cuando una expresión lambda se asigna a una variable de tipo Expression<TDelegate>, el compilador emite código para crear un árbol de expresión que represente la expresión lambda.

El compilador de C# puede generar árboles de expresión solo a partir de lambdas de expresión (o de lambdas de una sola línea). No pueden analizar lambdas de instrucción (o lambdas de varias líneas). Para obtener más información sobre las expresiones lambda en C#, consulte Expresiones lambda.

Los siguientes ejemplos de código muestran cómo crear un árbol de expresión que represente la expresión lambda `num => num < 5` con el compilador de C#.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Crear árboles de expresión mediante la API

Para crear árboles de expresión mediante la API, use la clase Expression. Esta clase contiene métodos de fábrica estáticos que crean nodos de árbol de expresión de tipos específicos, como, por ejemplo, ParameterExpression, que representa una variable o un parámetro, o MethodCallExpression, que representa una llamada a método. ParameterExpression, MethodCallExpression y los demás tipos específicos de expresión también se definen en el espacio de nombres System.Linq.Expressions. Estos tipos se derivan del tipo abstracto Expression.

En el siguiente ejemplo de código se muestra cómo crear un árbol de expresión que represente la expresión lambda `num => num < 5` mediante la API.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

En .NET Framework 4 y versiones posteriores, la API de árboles de expresión admite también asignaciones y expresiones de flujo de control como bucles, bloques condicionales y bloques `try-catch`. Con la API, se pueden crear árboles de expresión más complejos que los que pueden crear el compilador de C# a partir de expresiones lambda. En el siguiente ejemplo se indica cómo crear un árbol de expresión que calcula el factorial de un número.

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
        Expression.Loop(
    // Adding a conditional block into the loop.
           Expression.IfThenElse(
    // Condition: value > 1
               Expression.GreaterThan(value, Expression.Constant(1)),
    // If true: result *= value --
               Expression.MultiplyAssign(result,
                   Expression.PostDecrementAssign(value)),
    // If false, exit the loop and go to the label.
               Expression.Break(label, result)
           ),
    // Label to jump to.
       label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

Para obtener más información, consulte Generating Dynamic Methods with Expression Trees in Visual Studio 2010 (Generar métodos dinámicos con árboles de expresión en Visual Studio 2010), que también se aplica a las últimas versiones de Visual Studio.

Analizar árboles de expresión

En el siguiente ejemplo de código se muestra cómo la expresión del árbol que representa la expresión lambda `num => num < 5` se puede descomponer en partes.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Inmutabilidad de los árboles de expresión

Los árboles de expresión deben ser inmutables. Esto significa que, si desea modificar un árbol de expresión, deberá construir un nuevo árbol de expresión copiando el ya existente y reemplazando los nodos que hay en él. Puede usar un visitante de árbol de expresión para recorrer el árbol de expresión existente. Para obtener más información, consulte Procedimiento para modificar árboles de expresión (C#).

Compilar árboles de expresión

El tipo Expression<TDelegate> proporciona el método Compile que compila el código representado por un árbol de expresión en un delegado ejecutable.

En el siguiente ejemplo de código se indica cómo compilar un árbol de expresión y ejecutar el código resultante.

```
// Creating an expression tree.
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

Para obtener más información, consulte Procedimiento para ejecutar árboles de expresión (C#).

Vea también

- System.Linq.Expressions
- Procedimiento para ejecutar árboles de expresión (C#)
- Procedimiento para modificar árboles de expresión (C#)
- Expresiones lambda
- Información general sobre Dynamic Language Runtime
- Conceptos de programación (C#)

Procedimiento para ejecutar árboles de expresión (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo ejecutar un árbol de expresión. La ejecución de un árbol de expresión puede devolver un valor o simplemente realizar una acción, como llamar a un método.

Solo se pueden ejecutar los árboles de expresiones que representan expresiones lambda. Los árboles de expresiones que representan expresiones lambda son de tipo LambdaExpression o Expression<TDelegate>. Para ejecutar estos árboles de expresiones, llame al método Compile para crear un delegado ejecutable y, después, invoque el delegado.

> **NOTE**
> Si el tipo del delegado es desconocido, es decir, la expresión lambda es de tipo LambdaExpression y no Expression<TDelegate>, debe llamar al método DynamicInvoke en el delegado en lugar de invocarlo directamente.

Si un árbol de expresión no representa una expresión lambda, puede crear una nueva expresión lambda que tenga el árbol de expresión original como su cuerpo llamando al método Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>). Luego puede ejecutar la expresión lambda tal y como se ha descrito anteriormente en esta sección.

Ejemplo

En el ejemplo de código siguiente se muestra cómo ejecutar un árbol de expresión que representa la elevación de un número a una potencia mediante la creación de una expresión lambda y su posterior ejecución. Se muestra el resultado, que representa el número elevado a la potencia.

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2D), Expression.Constant(3D));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

Compilar el código

- Incluya el espacio de nombres System.Linq.Expressions.

Vea también

- Árboles de expresión (C#)
- Procedimiento para modificar árboles de expresión (C#)

Procedimiento para modificar árboles de expresión (C#)

18/03/2020 • 3 minutes to read • Edit Online

En este tema se muestra cómo modificar un árbol de expresión. Los árboles de expresiones son inmutables, lo que significa que no pueden modificarse directamente. Para cambiar un árbol de expresión, debe crear una copia de un árbol de expresión existente y, una vez creada la copia, realizar los cambios necesarios. Puede usar la clase ExpressionVisitor para recorrer un árbol de expresión existente y copiar cada nodo que visita.

Para modificar un árbol de expresión

1. Cree un nuevo proyecto de **Aplicación de consola**.

2. Agregue una directiva `using` al archivo para el espacio de nombres `System.Linq.Expressions`.

3. Agregue la clase `AndAlsoModifier` al proyecto.

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right, b.IsLiftedToNull,
b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

   Esta clase hereda la clase ExpressionVisitor y está especializada en la modificación de expresiones que representan operaciones `AND` condicionales. Cambia estas operaciones de una expresión `AND` condicional a una expresión `OR` condicional. Para ello, la clase invalida el método VisitBinary del tipo base, porque las expresiones `AND` condicionales se representan como expresiones binarias. En el método `VisitBinary`, si la expresión que se pasa representa una operación `AND` condicional, el código construye una nueva expresión que contiene el operador `OR` condicional en lugar del operador `AND` condicional. Si la expresión que se pasa a `VisitBinary` no representa una operación `AND` condicional, el método defiere a la implementación de la case base. Los métodos de clase base construyen nodos que son como los árboles de expresiones que se pasan, pero los subárboles de los nodos se reemplazan por los árboles de expresiones que genera de forma recursiva el visitante.

4. Agregue una directiva `using` al archivo para el espacio de nombres `System.Linq.Expressions`.

5. Agregue código al método `Main` en el archivo Program.cs para crear un árbol de expresión y pasarlo al método que lo modificará.

```
Expression<Func<string, bool>> expr = name => name.Length > 10 && name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression) expr);

Console.WriteLine(modifiedExpr);

/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

El código crea una expresión que contiene una operación `AND` condicional. Luego crea una instancia de la clase `AndAlsoModifier` y pasa la expresión al método `Modify` de esta clase. Se generan los árboles de expresiones tanto originales como modificados para mostrar el cambio.

6. Compile y ejecute la aplicación.

Vea también

- Procedimiento para ejecutar árboles de expresión (C#)
- Árboles de expresión (C#)

Procedimiento para usar árboles de expresión para crear consultas dinámicas (C#)

18/03/2020 • 5 minutes to read • Edit Online

En LINQ, los árboles de expresión se usan para representar consultas estructuradas destinadas a orígenes de datos que implementan IQueryable<T>. Por ejemplo, el proveedor LINQ implementa la interfaz IQueryable<T> para realizar consultas en almacenes de datos relacionales. El compilador de C# compila las consultas dirigidas a estos orígenes de datos en el código que genera un árbol de expresión en tiempo de ejecución. El proveedor de consultas puede después recorrer la estructura de datos del árbol de expresión y convertirla en un lenguaje de consulta adecuado para el origen de datos.

Los arboles de expresión también se usan en LINQ para representar expresiones lambda que se asignan a variables de tipo Expression<TDelegate>.

En este tema se describe cómo usar árboles de expresión para crear consultas LINQ dinámicas. Las consultas dinámicas son útiles cuando no se conocen los detalles de una consulta en tiempo de compilación. Por ejemplo, es posible que una aplicación proporcione una interfaz de usuario que permita al usuario final especificar uno o más predicados para filtrar los datos. Para poder usar LINQ para realizar consultas, este tipo de aplicación debe usar árboles de expresión para crear la consulta LINQ en tiempo de ejecución.

Ejemplo

En el ejemplo siguiente se muestra cómo usar árboles de expresión para crear una consulta en un origen de datos `IQueryable` y después ejecutarlo. El código crea un árbol de expresión para representar la consulta siguiente:

```
companies.Where(company => (company.ToLower() == "coho winery" || company.Length > 16))
         .OrderBy(company => company)
```

Los métodos de generador en el espacio de nombres System.Linq.Expressions se usan para crear árboles de expresión que representan las expresiones que constituyen la consulta global. Las expresiones que representan llamadas a los métodos de operador de consulta estándar hacen referencia a las implementaciones Queryable de estos métodos. El árbol de expresión final se pasa a la implementación CreateQuery<TElement>(Expression) del proveedor del origen de datos `IQueryable` para crear una consulta ejecutable de tipo `IQueryable`. Los resultados se obtienen enumerando esa variable de consulta.

```
// Add a using directive for System.Linq.Expressions.

string[] companies = { "Consolidated Messenger", "Alpine Ski House", "Southridge Video", "City Power & Light",
                   "Coho Winery", "Wide World Importers", "Graphic Design Institute", "Adventure Works",
                   "Humongous Insurance", "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
                   "Blue Yonder Airlines", "Trey Research", "The Phone Company",
                   "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee" };

// The IQueryable data to query.
IQueryable<String> queryableData = companies.AsQueryable<string>();

// Compose the expression tree that represents the parameter to the predicate.
ParameterExpression pe = Expression.Parameter(typeof(string), "company");

// ***** Where(company => (company.ToLower() == "coho winery" || company.Length > 16)) *****
// Create an expression tree that represents the expression 'company.ToLower() == "coho winery"'.
Expression left = Expression.Call(pe, typeof(string).GetMethod("ToLower", System.Type.EmptyTypes));
Expression right = Expression.Constant("coho winery");
```

```
Expression e1 = Expression.Equal(left, right);

// Create an expression tree that represents the expression 'company.Length > 16'.
left = Expression.Property(pe, typeof(string).GetProperty("Length"));
right = Expression.Constant(16, typeof(int));
Expression e2 = Expression.GreaterThan(left, right);

// Combine the expression trees to create an expression tree that represents the
// expression '(company.ToLower() == "coho winery" || company.Length > 16)'.
Expression predicateBody = Expression.OrElse(e1, e2);

// Create an expression tree that represents the expression
// 'queryableData.Where(company => (company.ToLower() == "coho winery" || company.Length > 16))'
MethodCallExpression whereCallExpression = Expression.Call(
    typeof(Queryable),
    "Where",
    new Type[] { queryableData.ElementType },
    queryableData.Expression,
    Expression.Lambda<Func<string, bool>>(predicateBody, new ParameterExpression[] { pe }));
// ***** End Where *****

// ***** OrderBy(company => company) *****
// Create an expression tree that represents the expression
// 'whereCallExpression.OrderBy(company => company)'
MethodCallExpression orderByCallExpression = Expression.Call(
    typeof(Queryable),
    "OrderBy",
    new Type[] { queryableData.ElementType, queryableData.ElementType },
    whereCallExpression,
    Expression.Lambda<Func<string, string>>(pe, new ParameterExpression[] { pe }));
// ***** End OrderBy *****

// Create an executable query from the expression tree.
IQueryable<string> results = queryableData.Provider.CreateQuery<string>(orderByCallExpression);

// Enumerate the results.
foreach (string company in results)
    Console.WriteLine(company);

/*  This code produces the following output:

    Blue Yonder Airlines
    City Power & Light
    Coho Winery
    Consolidated Messenger
    Graphic Design Institute
    Humongous Insurance
    Lucerne Publishing
    Northwind Traders
    The Phone Company
    Wide World Importers
*/
```

En este código se usa un número fijo de expresiones en el predicado que se pasa al método `Queryable.Where`. Pero se puede escribir una aplicación que combine un número variable de expresiones de predicado que dependa de la entrada del usuario. También se pueden variar los operadores de consulta estándar que se llaman en la consulta, dependiendo de la entrada del usuario.

Compilar el código

- Incluya el espacio de nombres System.Linq.Expressions.

Vea también

- Árboles de expresión (C#)
- Procedimiento para ejecutar árboles de expresión (C#)
- Especificar dinámicamente filtros con predicado en tiempo de ejecución

Depuración de árboles de expresión en Visual Studio (C#)

29/06/2020 • 2 minutes to read • Edit Online

Se puede analizar la estructura y el contenido de los árboles de expresión cuando se depuran las aplicaciones. Para obtener una introducción rápida a la estructura del árbol de expresión, puede usar la propiedad `DebugView`, que representa los árboles de expresión con una sintaxis especial. Observe que `DebugView` solo está disponible en modo de depuración.

Puesto que `DebugView` es una cadena, puede usar el visualizador de texto integrado para verla en varias líneas si selecciona **Visualizador de texto** en el icono de lupa situado junto a la etiqueta `DebugView`.

Como alternativa, puede instalar y usar un visualizador personalizado para árboles de expresión, como:

- Readable Expressions (licencia MIT, disponible en Visual Studio Marketplace), representa el árbol de expresión como código de C# con temas, con varias opciones de representación:

- Expression Tree Visualizer (licencia MIT), proporciona una vista de árbol del árbol de expresión y sus nodos individuales:

Para abrir un visualizador para un árbol de expresión

1. Haga clic en el icono de lupa que aparece junto al árbol de expresión en **Información sobre datos**, en una ventana **Inspección** o en las ventanas **Automático** o **Variables locales**.

   Se muestra una lista de los visualizadores disponibles:

2. Haga clic en el visualizador que desee usar.

Vea también

- Árboles de expresión (C#)
- Depurar en Visual Studio
- Create Custom Visualizers (Crear visualizadores personalizados)
- Sintaxis `DebugView`

Sintaxis de `DebugView`

18/03/2020 • 3 minutes to read • Edit Online

La propiedad `DebugView` (disponible solo durante la depuración) proporciona una representación de cadenas de árboles de expresión. La mayor parte de la sintaxis es bastante sencilla de entender; los casos especiales se describen en las siguientes secciones.

Cada ejemplo va seguido de un comentario del bloque, que contiene `DebugView`.

ParameterExpression

los nombres de variable System.Linq.Expressions.ParameterExpression se muestran con un símbolo `$` al principio.

Si un parámetro no tiene un nombre, se le asigna un nombre generado automáticamente, como `$var1` o `$var2`.

Ejemplos

```
ParameterExpression numParam =  Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam =  Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

Para los objetos System.Linq.Expressions.ConstantExpression que representan valores enteros, cadenas y `null`, se muestra el valor de la constante.

Para los tipos numéricos que tienen sufijos estándar como literales de C#, el sufijo se agrega al valor. En la tabla siguiente se muestran los sufijos asociados a varios tipos numéricos.

TIPO	PALABRA CLAVE	SUFIJO
System.UInt32	uint	U
System.Int64	long	L
System.UInt64	ulong	UL
System.Double	double	D
System.Single	float	F
System.Decimal	decimal	M

Ejemplos

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

Si el tipo de un objeto System.Linq.Expressions.BlockExpression difiere del tipo de la última expresión del bloque, el tipo se muestra entre corchetes angulares (`<` y `>`). De otro modo, el tipo del objeto BlockExpression no se muestra.

Ejemplos

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block =  Expression.Block(typeof(Object), Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

LambdaExpression

Los objetos System.Linq.Expressions.LambdaExpression se muestran junto con sus tipos delegados.

Si una expresión lambda no tiene un nombre, se le asigna un nombre generado automáticamente, como `#Lambda1` o `#Lambda2`.

Ejemplos

```
LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda =  Expression.Lambda<Func<int>>(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

Si especifica un valor predeterminado para el objeto System.Linq.Expressions.LabelExpression, este valor se muestra antes del objeto System.Linq.Expressions.LabelTarget.

El token `.Label` indica el inicio de la etiqueta. El token `.LabelTarget` indica el destino al que se va a saltar.

Si una etiqueta no tiene un nombre, se le asigna un nombre generado automáticamente, como `#Label1` o `#Label2`.

Ejemplos

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Operadores activados

Los operadores activados se muestran con el símbolo `#` delante del operador. Por ejemplo, el operador de adición activado se muestra como `#+`.

Ejemplos

```
Expression expr = Expression.AddChecked( Expression.Constant(1), Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0), typeof(int));
/*
    #(System.Int32)10D
*/
```

Iteradores (C#)

29/06/2020 • 12 minutes to read • Edit Online

Un *iterador* puede usarse para recorrer colecciones como listas y matrices.

Un método iterador o un descriptor de acceso `get` realiza una iteración personalizada en una colección. Un iterador usa la instrucción yield return para devolver cada elemento de uno en uno. Cuando se alcanza una instrucción `yield return`, se recuerda la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama a la función del iterador.

Para consumir un método iterador desde código de cliente, use una instrucción foreach o una consulta de LINQ.

En el ejemplo siguiente, la primera iteración del bucle `foreach` hace que continúe la ejecución del método de iterador `SomeNumbers` hasta que se alcance la primera instrucción `yield return`. Esta iteración devuelve un valor de 3, y la ubicación actual del método de iterador se conserva. En la siguiente iteración del bucle, la ejecución del método iterador continúa desde donde se dejó, deteniéndose de nuevo al alcanzar una instrucción `yield return`. Esta iteración devuelve un valor de 5, y la ubicación actual del método de iterador se vuelve a conservar. El bucle se completa al alcanzar el final del método iterador.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

El tipo de valor devuelto de un método de iterador o descriptor de acceso `get` puede ser IEnumerable, IEnumerable<T>, IEnumerator o IEnumerator<T>.

Puede usar una instrucción `yield break` para finalizar la iteración.

> **NOTE**
>
> En todos los ejemplos de este tema, excepto en el ejemplo de iterador simple, incluya directivas using para los espacios de nombres `System.Collections` y `System.Collections.Generic`.

Iterador simple

El ejemplo siguiente tiene una única instrucción `yield return` que está dentro de un bucle for. En `Main`, cada iteración del cuerpo de la instrucción `foreach` crea una llamada a la función de iterador, que continúa a la instrucción `yield return` siguiente.

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Creación de una clase de colección

En el ejemplo siguiente, la clase `DaysOfTheWeek` implementa la interfaz IEnumerable, que requiere un método GetEnumerator. El compilador llama implícitamente al método `GetEnumerator`, que devuelve un IEnumerator.

El método `GetEnumerator` devuelve las cadenas de una en una mediante la instrucción `yield return`.

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

En el ejemplo siguiente se crea una clase `Zoo` que contiene una colección de animales.

La instrucción `foreach` que hace referencia a la instancia de clase (`theZoo`) llama implícitamente al método `GetEnumerator`. Las instrucciones `foreach` que hacen referencia a las propiedades `Birds` y `Mammals` usan el método iterador con el nombre `AnimalsForType`.

```
static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods
```

```
    // Private methods.
    private IEnumerable AnimalsForType(Animal.TypeEnum type)
    {
        foreach (Animal theAnimal in animals)
        {
            if (theAnimal.Type == type)
            {
                yield return theAnimal.Name;
            }
        }
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}
```

Uso de iteradores con una lista genérica

En el ejemplo siguiente, la clase genérica Stack<T> implementa la interfaz genérica IEnumerable<T>. El método Push asigna valores a una matriz de tipo `T`. El método GetEnumerator devuelve los valores de la matriz con la instrucción `yield return`.

Además del método GetEnumerator genérico, el método GetEnumerator no genérico también debe implementarse. Esto es porque IEnumerable<T> se hereda de IEnumerable. La implementación no genérica aplaza la implementación genérica.

El ejemplo usa iteradores con nombre para admitir distintas formas de recorrer en iteración la misma colección de datos. Estos iteradores con nombre son las propiedades `TopToBottom` y `BottomToTop`, y el método `TopN`.

La propiedad `BottomToTop` usa un iterador en un descriptor de acceso `get`.

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    //  Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)
```

```
    foreach (int number in theStack.BottomToTop)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

        for (int index = top - 1; index >= startIndex; index--)
```

```
        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }

}
```

Información sobre la sintaxis

Un iterador se puede producir como un método o como un descriptor de acceso `get`. Un iterador no puede aparecer en un evento, un constructor de instancia, un constructor estático o un finalizador estático.

Debe existir una conversión implícita desde el tipo de expresión en la instrucción `yield return` al argumento de tipo para el valor `IEnumerable<T>` devuelto por el iterador.

En C#, un método iterador no puede tener ningún parámetro `in`, `ref` o `out`.

En C#, `yield` no es una palabra reservada y solo tiene un significado especial cuando se usa antes de una palabra clave `return` o `break`.

Implementación técnica

Aunque un iterador se escribe como un método, el compilador lo traduce a una clase anidada que es, en realidad, una máquina de estados. Esta clase realiza el seguimiento de la posición del iterador mientras el bucle `foreach` continúe en el código de cliente.

Para ver lo que hace el compilador, puede usar la herramienta Ildasm.exe para ver el código de lenguaje intermedio de Microsoft que se genera para un método de iterador.

Cuando crea un iterador para una clase o struct, no necesita implementar la interfaz IEnumerator completa. Cuando el compilador detecta el iterador, genera automáticamente los métodos `Current`, `MoveNext` y `Dispose` de la interfaz IEnumerator o IEnumerator<T>.

En cada iteración sucesiva del bucle `foreach` (o la llamada directa a `IEnumerator.MoveNext`), el cuerpo de código del iterador siguiente se reanuda después de la instrucción `yield return` anterior. Después continúa con la siguiente instrucción `yield return` hasta que se alcanza el final del cuerpo del iterador, o hasta que se encuentra una instrucción `yield break`.

Los iteradores no admiten el método IEnumerator.Reset. Para volver a recorrer en iteración desde el principio, se debe obtener un nuevo iterador. Una llamada a Reset en el iterador devuelto por un método de iterador inicia una excepción NotSupportedException.

Para obtener más información, vea la Especificación del lenguaje C#.

Uso de iteradores

Los iteradores permiten mantener la simplicidad de un bucle `foreach` cuando se necesita usar código complejo para rellenar una secuencia de lista. Esto puede ser útil si quiere hacer lo siguiente:

- Modificar la secuencia de lista después de la primera iteración del bucle `foreach`.
- Evitar que se cargue totalmente una lista grande antes de la primera iteración de un bucle `foreach`. Un ejemplo es una búsqueda paginada para cargar un lote de filas de tabla. Otro ejemplo es el método EnumerateFiles, que implementa iteradores en .NET.
- Encapsular la creación de la lista en el iterador. En el método iterador, puede crear la lista y después devolver cada resultado en un bucle.

Vea también

- System.Collections.Generic
- IEnumerable<T>
- foreach, in
- yield
- Utilizar foreach con matrices
- Genéricos

Language-Integrated Query (LINQ)

20/04/2020 • 7 minutes to read • Edit Online

Language-Integrated Query (LINQ) es el nombre de un conjunto de tecnologías basadas en la integración de capacidades de consulta directamente en el lenguaje C#. Tradicionalmente, las consultas con datos se expresaban como cadenas simples sin comprobación de tipos en tiempo de compilación ni compatibilidad con IntelliSense. Además, tendrá que aprender un lenguaje de consulta diferente para cada tipo de origen de datos: bases de datos SQL, documentos XML, varios servicios web y así sucesivamente. Con LINQ, una consulta es una construcción de lenguaje de primera clase, como clases, métodos y eventos. Escribe consultas en colecciones de objetos fuertemente tipadas con palabras clave del lenguaje y operadores familiares. La familia de tecnologías de LINQ proporciona una experiencia de consulta coherente para objetos (LINQ to Objects), bases de datos relacionales (LINQ to SQL) y XML (LINQ to XML).

Para un desarrollador que escribe consultas, la parte más visible de "lenguaje integrado" de LINQ es la expresión de consulta. Las expresiones de consulta se escriben con una *sintaxis de consulta* declarativa. Con la sintaxis de consulta, puede realizar operaciones de filtrado, ordenación y agrupamiento en orígenes de datos con el mínimo código. Utilice los mismos patrones de expresión de consulta básica para consultar y transformar datos de bases de datos SQL, conjuntos de datos de ADO .NET, secuencias y documentos XML y colecciones. NET.

Puede escribir consultas LINQ en C# para bases de datos de SQL Server, documentos XML, conjuntos de datos ADO.NET y cualquier colección de objetos que admita IEnumerable o la interfaz genérica IEnumerable<T>. La compatibilidad con LINQ también se proporciona por terceros para muchos servicios web y otras implementaciones de base de datos.

En el ejemplo siguiente se muestra la operación de consulta completa. La operación completa incluye crear un origen de datos, definir la expresión de consulta y ejecutar la consulta en una instrucción `foreach`.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

En la siguiente ilustración de Visual Studio se muestra una consulta LINQ parcialmente completa en una base de datos de SQL Server en C# y Visual Basic con la comprobación de tipos completa y la compatibilidad con IntelliSense:

Información general sobre la expresión de consulta

- Las expresiones de consulta se pueden utilizar para consultar y transformar los datos de cualquier origen de datos habilitado para LINQ. Por ejemplo, una sola consulta puede recuperar datos de una base de datos SQL y generar una secuencia XML como salida.
- Las expresiones de consulta son fáciles de controlar porque utilizan muchas construcciones de lenguaje C# familiares.
- Todas las variables de una expresión de consulta están fuertemente tipadas, aunque en muchos casos no es necesario proporcionar el tipo explícitamente porque el compilador puede deducirlo. Para más información, vea Relaciones entre tipos en operaciones de consulta LINQ.
- Una consulta no se ejecuta hasta que no se realiza la iteración a través de la variable de consulta, por ejemplo, en una instrucción `foreach`. Para más información, vea Introducción a las consultas LINQ.
- En tiempo de compilación, las expresiones de consulta se convierten en llamadas al método de operador de consulta estándar según las reglas establecidas en la especificación de C#. Cualquier consulta que se puede expresar con sintaxis de consulta también se puede expresar mediante sintaxis de método. Sin embargo, en la mayoría de los casos, la sintaxis de consulta es más legible y concisa. Para más información, vea Especificación del lenguaje C# e Información general sobre operadores de consulta estándar.
- Como regla al escribir consultas LINQ, se recomienda utilizar la sintaxis de consulta siempre que sea posible y la sintaxis de método cuando sea necesario. No hay diferencias semánticas ni de rendimiento entre estas dos formas diversas. Las expresiones de consulta suelen ser más legibles que las expresiones equivalentes escritas con la sintaxis de método.
- Algunas operaciones de consulta, como Count o Max, no tienen ninguna cláusula de expresión de consulta equivalente, de modo que deben expresarse como una llamada de método. La sintaxis de método se puede combinar con la sintaxis de consulta de varias maneras. Para más información, vea Sintaxis de consultas y sintaxis de métodos en LINQ.
- Las expresiones de consulta pueden compilarse en árboles de expresión o en delegados, en función del tipo al que se aplica la consulta. Las consultas IEnumerable<T> se compilan en delegados. Las consultas IQueryable y IQueryable<T> se compilan en árboles de expresión. Para más información, vea Árboles de expresión.

Pasos siguientes

Para obtener más información sobre LINQ, empiece a familiarizarse con algunos conceptos básicos en Conceptos básicos de las expresiones de consultas y, después, lea la documentación de la tecnología de LINQ en la que esté interesado:

- Documentos XML: LINQ to XML
- ADO.NET Entity Framework: LINQ to Entities
- Colecciones .NET, archivos y cadenas, entre otros: LINQ to Objects

Para comprender mejor los aspectos generales de LINQ, vea LINQ in C# (LINQ en C#).

Para empezar a trabajar con LINQ en C#, vea el tutorial Trabajar con LINQ.

Introducción a las consultas LINQ (C#)

22/04/2020 • 11 minutes to read • Edit Online

Una *consulta* es una expresión que recupera datos de un origen de datos. Las consultas se suelen expresar en un lenguaje de consultas especializado. Con el tiempo se han desarrollado diferentes lenguajes para los distintos tipos de orígenes de datos, como SQL para las bases de datos relacionales y XQuery para XML. Por lo tanto, los programadores han tenido que aprender un lenguaje de consultas nuevo para cada tipo de origen de datos o formato de datos que deben admitir. LINQ simplifica esta situación al ofrecer un modelo coherente para trabajar con los datos de varios formatos y orígenes. En una consulta LINQ siempre se trabaja con objetos. Se usan los mismos patrones de codificación básicos para consultar y transformar datos de documentos XML, bases de datos SQL, conjuntos de datos de ADO.NET, colecciones de .NET y cualquier otro formato para el que haya disponible un proveedor de LINQ.

Las tres partes de una operación de consulta

Todas las operaciones de consulta LINQ constan de tres acciones distintas:

1. Obtener el origen de datos.
2. Crear la consulta.
3. Ejecutar la consulta.

En el siguiente ejemplo se muestra cómo se expresan las tres partes de una operación de consulta en código fuente. En el ejemplo se usa una matriz de enteros como origen de datos para su comodidad, aunque se aplican los mismos conceptos a otros orígenes de datos. En el resto de este tema se hará referencia a este ejemplo.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.Write("{0,1} ", num);
        }
    }
}
```

En la siguiente ilustración se muestra toda la operación de consulta. En LINQ, la ejecución de la consulta es distinta de la propia consulta. En otras palabras, no ha recuperado ningún dato simplemente creando una variable de consulta.

El origen de datos

En el ejemplo anterior, como el origen de datos es una matriz, admite implícitamente la interfaz genérica IEnumerable<T>. Este hecho implica que se puede consultar con LINQ. Se ejecuta una consulta en una instrucción `foreach`, y `foreach` requiere IEnumerable o bien IEnumerable<T>. Los tipos compatibles con IEnumerable<T> o una interfaz derivada, como la interfaz genérica IQueryable<T>, se denominan *tipos consultables*.

Un tipo consultable no requiere ninguna modificación ni ningún tratamiento especial para actuar como origen de datos de LINQ. Si el origen de datos no está en la memoria como tipo consultable, el proveedor de LINQ debe representarlo como tal. Por ejemplo, LINQ to XML carga un documento XML en un tipo XElement consultable:

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

Con LINQ to SQL, primero se crea una asignación relacional de objetos en tiempo de diseño ya sea manualmente o mediante las herramientas de LINQ to SQL en Visual Studio. Después, se escriben las consultas en los objetos y, en tiempo de ejecución, LINQ to SQL controla la comunicación con la base de datos. En el ejemplo siguiente, `Customers` representa una tabla específica en una base de datos, y el tipo del resultado de la consulta, IQueryable<T>, se deriva de IEnumerable<T>.

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

Para obtener más información sobre cómo crear tipos específicos de orígenes de datos, consulte la documentación de los distintos proveedores de LINQ. Aun así, la regla básica es muy sencilla: un origen de datos de LINQ es cualquier objeto que admita la interfaz genérica IEnumerable<T> o una interfaz que la haya heredado.

> **NOTE**
> Los tipos como ArrayList, que admiten la interfaz no genérica IEnumerable, también se pueden usar como origen de datos de LINQ. Para más información, consulte el procedimiento para consultar un objeto ArrayList con LINQ (C#).

La consulta

La consulta especifica la información que se debe recuperar de los orígenes de datos. Opcionalmente, una consulta también especifica cómo se debe ordenar, agrupar y conformar esa información antes de que se devuelva. Las consultas se almacenan en una variable de consulta y se inicializan con una expresión de consulta. Para facilitar la escritura de consultas, C# ha incorporado una nueva sintaxis de consulta.

La consulta del ejemplo anterior devuelve todos los números pares de la matriz de enteros. La expresión de consulta contiene tres cláusulas: `from`, `where` y `select` (si está familiarizado con SQL, habrá observado que el orden de las cláusulas se invierte respecto al orden de SQL). La cláusula `from` especifica el origen de datos, la cláusula `where` aplica el filtro y la cláusula `select` especifica el tipo de los elementos devueltos. Estas y otras cláusulas de consulta se tratan con detalle en la sección Language Integrated Query (LINQ). Por ahora, lo importante es que en LINQ la variable de consulta no efectúa ninguna acción y no devuelve ningún dato. Lo único que hace es almacenar la información necesaria para generar los resultados cuando se ejecuta la consulta en algún momento posterior. Para obtener más información sobre cómo se construyen las consultas en segundo plano, vea Información general sobre operadores de consulta estándar (C#).

> **NOTE**
> Las consultas también se pueden expresar empleando una sintaxis de método. Para obtener más información, vea Query Syntax and Method Syntax in LINQ (Sintaxis de consulta y sintaxis de método en LINQ).

Ejecución de la consulta

Ejecución aplazada

Como se ha indicado anteriormente, la variable de consulta solo almacena los comandos de consulta. La ejecución real de la consulta se aplaza hasta que se procese una iteración en la variable de consulta en una instrucción `foreach`. Este concepto se conoce como *ejecución aplazada* y se muestra en el ejemplo siguiente:

```
//  Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

La instrucción `foreach` es también donde se recuperan los resultados de la consulta. Por ejemplo, en la consulta anterior, la variable de iteración `num` contiene cada valor (de uno en uno) en la secuencia devuelta.

Dado que la propia variable de consulta nunca contiene los resultados de la consulta, puede ejecutarla tantas veces como desee. Por ejemplo, se puede tener una base de datos que esté siendo actualizada de forma continua por otra aplicación. En su aplicación, se puede crear una consulta que recupere los datos más recientes y se puede ejecutar repetidamente de acuerdo con un intervalo para recuperar resultados diferentes cada vez.

Forzar la ejecución inmediata

Las consultas que llevan a cabo funciones de agregación en un intervalo de elementos de origen primero deben recorrer en iteración dichos elementos. Ejemplos de estas consultas son `Count`, `Max`, `Average` y `First`. Se ejecutan sin una instrucción `foreach` explícita, ya que la propia consulta debe usar `foreach` para poder

devolver un resultado. Tenga en cuenta también que estos tipos de consultas devuelven un único valor, y no una colección `IEnumerable`. La consulta siguiente devuelve un recuento de los números pares de la matriz de origen:

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

Para forzar la ejecución inmediata de cualquier consulta y almacenar en caché los resultados correspondientes, puede llamar a los métodos ToList o ToArray.

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

También puede forzar la ejecución colocando el bucle `foreach` justo después de la expresión de consulta, aunque, si se llama a `ToList` o a `ToArray`, también se almacenan en caché todos los datos de un objeto de colección.

Vea también

- Introducción a LINQ en C#
- Tutorial: Escribir consultas en C#
- Language-Integrated Query (LINQ)
- foreach, in
- Palabras clave para consultas (LINQ)

LINQ y tipos genéricos (C#)

29/06/2020 • 3 minutes to read • Edit Online

Las consultas LINQ se basan en tipos genéricos incorporados en la versión 2.0 de .NET Framework. No es necesario tener conocimientos avanzados de genéricos para poder empezar a escribir consultas, aunque debería entender dos conceptos básicos:

1. Al crear una instancia de una clase de colección genérica como List<T>, reemplace la "T" por el tipo de objetos que contendrá la lista. Por ejemplo, una lista de cadenas se expresa como `List<string>` y una lista de objetos `Customer` se expresa como `List<Customer>`. Las listas genéricas están fuertemente tipadas y ofrecen muchas ventajas respecto a las colecciones que almacenan sus elementos como Object. Si intenta agregar un `Customer` a una `List<string>`, se producirá un error en tiempo de compilación. Usar colecciones genéricas es fácil porque no es necesario efectuar ninguna conversión de tipos en tiempo de ejecución.

2. IEnumerable<T> es la interfaz que permite enumerar las clases de colección genéricas mediante la instrucción `foreach`. Las clases de colección genéricas admiten IEnumerable<T> simplemente como clases de colección no genéricas como ArrayList admite IEnumerable.

Para obtener más información sobre los genéricos, vea Genéricos.

Variables IEnumerable<T> en las consultas LINQ

Las variables de consulta LINQ tienen el tipo IEnumerable<T> o un tipo derivado como IQueryable<T>. Cuando vea una variable de consulta que tiene el tipo `IEnumerable<Customer>`, significa que, al ejecutarse, la consulta generará una secuencia de cero o más objetos `Customer`.

```
IEnumerable<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

Para obtener más información, vea Relaciones entre tipos en operaciones de consulta LINQ.

Permitir que el compilador controle las declaraciones de tipo genérico

Si lo prefiere, puede evitar la sintaxis genérica mediante la palabra clave var. La palabra clave `var` indica al compilador que infiera el tipo de una variable de consulta examinando el origen de datos especificado en la cláusula `from`. En el ejemplo siguiente se genera el mismo código compilado que en el ejemplo anterior:

```
var customerQuery2 =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

La palabra clave `var` es útil cuando el tipo de la variable es obvio o cuando no es tan importante especificar explícitamente los tipos genéricos anidados, como los que generan las consultas de grupo. Le recordamos que, si usa `var`, debe tener presente que puede dificultar la lectura del código a otros usuarios. Para más información, vea Variables locales con asignación implícita de tipos.

Vea también

- Genéricos

Operaciones básicas de consulta LINQ (C#)

18/03/2020 • 9 minutes to read • Edit Online

En este tema se ofrece una breve introducción a las expresiones de consulta LINQ y algunas de las clases de operaciones típicas que se realizan en una consulta. En los temas siguientes se ofrece información más detallada:

Expresiones de consulta LINQ

Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))

Walkthrough: Writing Queries in C# (Tutorial: Escribir consultas en C#)

> **NOTE**
> Si ya está familiarizado con un lenguaje de consultas como SQL o XQuery, puede omitir la mayoría de este tema. Lea la parte dedicada a la "cláusula `from`" en la sección siguiente para obtener información sobre el orden de las cláusulas en las expresiones de consulta LINQ.

Obtener un origen de datos

En una consulta LINQ, el primer paso es especificar el origen de datos. En C#, como en la mayoría de los lenguajes de programación, se debe declarar una variable antes de poder usarla. En una consulta LINQ, la cláusula `from` aparece en primer lugar para introducir el origen de datos (`customers`) y la *variable de rango* (`cust`).

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

La variable de rango es como la variable de iteración en un bucle `foreach`, con la diferencia de que en una expresión de consulta realmente no se produce ninguna iteración. Cuando se ejecuta la consulta, la variable de rango actúa como referencia para cada elemento sucesivo de `customers`. Dado que el compilador puede deducir el tipo de `cust`, no tiene que especificarlo explícitamente. Una cláusula `let` puede introducir variables de rango adicionales. Para obtener más información, vea let (Cláusula).

> **NOTE**
> Para los orígenes de datos no genéricos, como ArrayList, el tipo de la variable de rango debe establecerse explícitamente. Para más información, consulte el procedimiento para consultar un objeto ArrayList con LINQ (C#) y Cláusula from.

Filtros

Probablemente la operación de consulta más común es aplicar un filtro en forma de expresión booleana. El filtro hace que la consulta devuelva solo los elementos para los que la expresión es verdadera. El resultado se genera mediante la cláusula `where`. El filtro aplicado especifica qué elementos se deben excluir de la secuencia de origen. En el ejemplo siguiente, solo se devuelven los `customers` cuya dirección se encuentra en Londres.

```
var queryLondonCustomers = from cust in customers
                           where cust.City == "London"
                           select cust;
```

Puede usar los operadores lógicos `AND` y `OR` de C#, con los que ya estará familiarizado, para aplicar las expresiones de filtro que sean necesarias en la cláusula `where`. Por ejemplo, para devolver solo los clientes con dirección en "London" `AND` cuyo nombre sea "Devon", escribiría el código siguiente:

```
where cust.City == "London" && cust.Name == "Devon"
```

Para devolver los clientes con dirección en Londres o París, escribiría el código siguiente:

```
where cust.City == "London" || cust.City == "Paris"
```

Para obtener más información, vea where (Cláusula).

Ordenación

A menudo es necesario ordenar los datos devueltos. La cláusula `orderby` hará que los elementos de la secuencia devuelta se ordenen según el comparador predeterminado del tipo que se va a ordenar. Por ejemplo, la consulta siguiente se puede extender para ordenar los resultados según la propiedad `Name`. Dado que `Name` es una cadena, el comparador predeterminado realiza una ordenación alfabética de la A a la Z.

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Para ordenar los resultados en orden inverso, de la Z a la A, use la cláusula `orderby…descending`.

Para obtener más información, vea orderby (Cláusula).

Agrupar

La cláusula `group` permite agrupar los resultados según la clave que se especifique. Por ejemplo, podría especificar que los resultados se agrupen por `City` para que todos los clientes de London o París estén en grupos individuales. En este caso, la clave es `cust.City`.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
  var queryCustomersByCity =
      from cust in customers
      group cust by cust.City;

  // customerGroup is an IGrouping<string, Customer>
  foreach (var customerGroup in queryCustomersByCity)
  {
      Console.WriteLine(customerGroup.Key);
      foreach (Customer customer in customerGroup)
      {
          Console.WriteLine("    {0}", customer.Name);
      }
  }
```

Al finalizar una consulta con una cláusula `group`, los resultados adoptan la forma de una lista de listas. Cada elemento de la lista es un objeto que tiene un miembro `Key` y una lista de elementos agrupados bajo esa clave. Al procesar una iteración en una consulta que genera una secuencia de grupos, debe usar un bucle `foreach` anidado. El bucle exterior recorre en iteración cada grupo y el bucle interior recorre en iteración los miembros de cada grupo.

Si debe hacer referencia a los resultados de una operación de grupo, puede usar la palabra clave `into` para crear un identificador con el que se puedan realizar más consultas. La consulta siguiente devuelve solo los grupos que contienen más de dos clientes:

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

Para obtener más información, vea group (Cláusula).

Combinación

Las operaciones de combinación crean asociaciones entre las secuencias que no se modelan explícitamente en los orígenes de datos. Por ejemplo, puede realizar una combinación para buscar todos los clientes y distribuidores que tengan la misma ubicación. En LINQ, la cláusula `join` funciona siempre con colecciones de objetos, en lugar de con tablas de base de datos directamente.

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

En LINQ no es necesario usar `join` tan a menudo como en SQL, porque las claves externas en LINQ se representan en el modelo de objetos como propiedades que contienen una colección de elementos. Por ejemplo, un objeto `Customer` contiene una colección de objetos `Order`. En lugar de realizar una combinación, tiene acceso a los pedidos usando la notación de punto:

```
from order in Customer.Orders...
```

Para obtener más información, vea join (Cláusula, Referencia de C#).

Selección (proyecciones)

La cláusula `select` genera resultados de consulta y especifica la "forma" o el tipo de cada elemento devuelto. Por ejemplo, puede especificar si sus resultados estarán compuestos de objetos `Customer` completos, un solo miembro, un subconjunto de miembros o algún tipo de resultado completamente diferente basado en un cálculo o en un objeto nuevo. Cuando la cláusula `select` genera algo distinto de una copia del elemento de origen, la operación se denomina *proyección*. El uso de proyecciones para transformar los datos es una función eficaz de las expresiones de consulta LINQ. Para obtener más información, vea Transformaciones de datos con LINQ (C#) y select (cláusula).

Vea también

- Expresiones de consulta LINQ
- Walkthrough: Writing Queries in C# (Tutorial: Escribir consultas en C#)
- Palabras clave para consultas (LINQ)
- Tipos anónimos

Transformaciones de datos con LINQ (C#)

29/06/2020 • 9 minutes to read • Edit Online

Language-Integrated Query (LINQ) no solo es para recuperar datos. También es una herramienta eficaz para transformarlos. Mediante el uso de un consulta LINQ, se puede usar una secuencia de origen como entrada y modificarla de muchas maneras para crear una nueva secuencia de salida. Por medio de ordenaciones y agrupaciones se puede modificar la propia secuencia sin modificar los elementos. Pero quizás la característica más eficaz de las consultas LINQ es la capacidad para crear nuevos tipos. Esto se realiza en la cláusula select. Por ejemplo, puede realizar las tareas siguientes:

- Combinar varias secuencias de entrada en una sola secuencia de salida que tiene un tipo nuevo.
- Crear secuencias de salida cuyos elementos estén formados por una o varias propiedades de cada elemento de la secuencia de origen.
- Crear secuencias de salida cuyos elementos estén formados por los resultados de las operaciones realizadas en el origen de datos.
- Crear secuencias de salida en un formato diferente. Por ejemplo, se pueden transformar datos de filas de SQL o archivos de texto en XML.

Estos son solo algunos ejemplos. Por supuesto, estas transformaciones pueden combinarse de diversas formas en la misma consulta. Además, se puede usar la secuencia de salida de una consulta como la secuencia de entrada para una nueva consulta.

Combinar varias entradas en una secuencia de salida

Se puede usar una consulta LINQ para crear una secuencia de salida que contiene los elementos de más de una secuencia de entrada. En el ejemplo siguiente se muestra cómo combinar dos estructuras de datos en memoria, pero se pueden aplicar los mismos principios para combinar datos de XML o de orígenes SQL o DataSet. Supongamos los dos tipos de clase siguientes:

```
class Student
{
    public string First { get; set; }
    public string Last {get; set;}
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

En el ejemplo siguiente se muestra la consulta:

```
class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                Last="Omelchenko",
                ID=111,
                Street="123 Main Street",
                City="Seattle",
                Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                Last="O'Donnell",
                ID=112,
                Street="124 Main Street",
                City="Redmond",
                Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                Last="Mortensen",
                ID=113,
                Street="125 Main Street",
                City="Lake City",
                Scores= new List<int> { 88, 94, 65, 91 } },
        };

        // Create the second data source.
        List<Teacher> teachers = new List<Teacher>()
        {
            new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle" },
            new Teacher { First="Alex", Last="Robinson", ID=956, City="Redmond" },
            new Teacher { First="Michiyo", Last="Sato", ID=972, City="Tacoma" }
        };

        // Create the query.
        var peopleInSeattle = (from student in students
                    where student.City == "Seattle"
                    select student.Last)
                    .Concat(from teacher in teachers
                            where teacher.City == "Seattle"
                            select teacher.Last);

        Console.WriteLine("The following students and teachers live in Seattle:");
        // Execute the query.
        foreach (var person in peopleInSeattle)
        {
            Console.WriteLine(person);
        }

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    The following students and teachers live in Seattle:
    Omelchenko
    Beebe
 */
```

Para obtener más información, vea join (Cláusula) y select (Cláusula).

Seleccionar un subconjunto de cada elemento de origen

Hay dos maneras principales de seleccionar un subconjunto de cada elemento de la secuencia de origen:

1. Para seleccionar a un solo miembro del elemento de origen, use la operación de punto. En el ejemplo siguiente, suponga que un objeto `Customer` contiene varias propiedades públicas, incluida una cadena denominada `City`. Cuando se ejecuta, esta consulta genera una secuencia de salida de cadenas.

```
var query = from cust in Customers
            select cust.City;
```

2. Para crear elementos que contengan más de una propiedad del elemento de origen, se puede usar un inicializador de objeto con un objeto con nombre o un tipo anónimo. En el ejemplo siguiente se muestra el uso de un tipo anónimo para encapsular dos propiedades de cada elemento `Customer`:

```
var query = from cust in Customer
            select new {Name = cust.Name, City = cust.City};
```

Para obtener más información, vea Inicializadores de objeto y de colección y Tipos anónimos.

Transformar objetos en memoria en XML

Las consultas LINQ facilitan la transformación de datos entre estructuras de datos en memoria, bases de datos SQL, conjuntos de datos de ADO.NET y documentos o secuencias de XML. En el siguiente ejemplo se transforman objetos de una estructura de datos en memoria en elementos XML.

```
class XMLTransform
{
    static void Main()
    {
        // Create the data source by using a collection initializer.
        // The Student class was defined previously in this topic.
        List<Student> students = new List<Student>()
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores = new List<int>{97, 92, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores = new List<int>{75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores = new List<int>{88, 94, 65, 91}},
        };

        // Create the query.
        var studentsToXML = new XElement("Root",
            from student in students
            let scores = string.Join(",", student.Scores)
            select new XElement("student",
                       new XElement("First", student.First),
                       new XElement("Last", student.Last),
                       new XElement("Scores", scores)
                   ) // end "student"
               ); // end "Root"

        // Execute the query.
        Console.WriteLine(studentsToXML);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

El código produce el siguiente resultado de XML:

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>
```

Para obtener más información, vea Creating XML Trees (C#) (Creación de árboles XML [C#]).

Realizar operaciones en los elementos de origen

Es posible que una secuencia de salida no contenga ningún elemento o propiedades de elemento de la secuencia de origen. En su lugar, es posible que la salida sea una secuencia de valores que se calcula usando los elementos de origen como argumentos de entrada.

La consulta siguiente tomará una secuencia de números que representan radios de círculos, calculará el área de cada radio y devolverá una secuencia de salida que contenga cadenas con el formato del área calculada.

Se dará formato a cada cadena de la secuencia de salida mediante la interpolación de cadenas. Una cadena interpolada presentará un elemento `$` delante de las comillas de apertura de la cadena, y las operaciones se pueden realizar entre llaves colocadas dentro de la cadena interpolada. Tras realizar las operaciones, se concatenarán los resultados.

> **NOTE**
>
> No se admite llamar a métodos en expresiones de consulta si la consulta se va a convertir a otro dominio. Por ejemplo, no se puede llamar a un método normal de C# en LINQ to SQL porque SQL Server no tiene contexto para él. En cambio, se pueden asignar procedimientos almacenados a los métodos y llamar a los primeros. Para obtener más información, vea Procedimientos almacenados.

```
class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' = {r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        IEnumerable<string> output =
            from rad in radii
            select $"Area = {rad * rad * Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area = 3.14
    Area = 12.57
    Area = 28.27
*/
```

Vea también

- Language Integrated Query (LINQ) (C#)
- LINQ to SQL
- LINQ to DataSet
- LINQ to XML (C#)
- Expresiones de consulta LINQ
- select (cláusula)

Relaciones entre tipos en operaciones de consulta LINQ (C#)

18/03/2020 • 5 minutes to read • Edit Online

Para escribir las consultas eficazmente, es necesario comprender cómo los tipos de las variables en una operación de consulta completa se relacionan entre sí. Si entiende estas relaciones comprenderá más fácilmente los ejemplos de LINQ y los ejemplos de código de la documentación. Además, entenderá lo que sucede en segundo plano cuando los tipos de las variables se declaran implícitamente mediante `var`.

las operaciones de consulta LINQ tienen un establecimiento fuertemente tipado en el origen de datos, en la propia consulta y en la ejecución de la consulta. El tipo de las variables de la consulta debe ser compatible con el tipo de los elementos del origen de datos y con el tipo de la variable de iteración de la instrucción `foreach`. Este establecimiento inflexible de tipos garantiza que los errores de tipos se detectan en tiempo de compilación, cuando aún se pueden corregir antes de que los usuarios los detecten.

Para mostrar estas relaciones de tipos, en la mayoría de los ejemplos siguientes se usan tipos explícitos para todas las variables. En el último ejemplo se muestra cómo se aplican los mismos principios incluso al usar tipos implícitos mediante var.

Consultas que no transforman los datos de origen

La ilustración siguiente muestra una operación de consulta de LINQ to Objects que no realiza ninguna transformación de los datos. El origen contiene una secuencia de cadenas y el resultado de la consulta también es una secuencia de cadenas.

1. El argumento de tipo del origen de datos determina el tipo de la variable de rango.
2. El tipo del objeto que está seleccionado determina el tipo de la variable de consulta. Aquí, `name` es una cadena. Por tanto, la variable de consulta es `IEnumerable<string>`.
3. La variable de consulta se procesa en iteración en la instrucción `foreach`. Dado que la variable de consulta es una secuencia de cadenas, la variable de iteración también es una cadena.

Consultas que transforman los datos de origen

En la ilustración siguiente se muestra una operación de consulta LINQ to SQL que realiza una transformación simple de los datos. La consulta usa una secuencia de objetos `Customer` como entrada y selecciona solo la propiedad `Name` en el resultado. Dado que `Name` es una cadena, la consulta genera una secuencia de cadenas como resultado.

```
Table<Customer> Customers = db.GetTable<Customers>();

IQueryable<string> custNameQuery =
                    from cust in Customers
                       where cust.City == "London"
                       select cust.Name;

foreach (string str in custNameQuery)
{
     Console.WriteLine(str);
}
```

1. El argumento de tipo del origen de datos determina el tipo de la variable de rango.
2. La instrucción `select` devuelve la propiedad `Name` en lugar del objeto `Customer` completo. Dado que `Name` es una cadena, el argumento de tipo de `custNameQuery` es `string`, no `Customer`.
3. Dado que `custNameQuery` es una secuencia de cadenas, la variable de iteración del bucle `foreach` también debe ser `string`.

En la ilustración siguiente se muestra una transformación un poco más compleja. La instrucción `select` devuelve un tipo anónimo que captura solo dos miembros del objeto `Customer` original.

```
Table<Customer> Customers = db.GetTable<Customers>();

var namePhoneQuery =
                 from cust in Customers
                   where cust.City == "London"
                   select new { name = cust.Name,
                        phone = cust.Phone };

foreach (var item in namePhoneQuery)
{
     Console.WriteLine(item);
}
```

1. El argumento de tipo del origen de datos siempre es el tipo de la variable de rango de la consulta.
2. Dado que la instrucción `select` genera un tipo anónimo, la variable de consulta debe declararse implícitamente mediante `var`.
3. Dado que el tipo de la variable de consulta es implícito, la variable de iteración del bucle `foreach` también debe ser implícita.

Permitir que el compilador deduzca la información de tipo

Aunque debería comprender las relaciones de los tipos en una operación de consulta, tiene la opción de que el compilador le haga todo el trabajo. La palabra clave var se puede usar para cualquier variable local en una operación de consulta. La ilustración siguiente es similar al ejemplo número 2 que se ha analizado anteriormente. En cambio, el compilador proporciona el tipo seguro de cada variable en la operación de consulta.

```
var Customers = db.GetTable<Customers>();

var custQuery = from cust in Customers
                where cust.City == "London"
                select cust;

foreach (var item in custQuery)
{
    Console.WriteLine(item);
}
```

Para obtener más información sobre `var`, vea Variables locales con asignación implícita de tipos.

Sintaxis de consultas y sintaxis de métodos en LINQ (C#)

16/04/2020 • 9 minutes to read • Edit Online

La mayoría de las consultas de la documentación introductoria de Language Integrated Query (LINK) se escribe con la sintaxis de consulta declarativa de LINQ. Pero la sintaxis de consulta debe traducirse en llamadas de método para .NET Common Language Runtime (CLR) al compilar el código. Estas llamadas de método invocan los operadores de consulta estándar, que tienen nombres tales como `Where`, `Select`, `GroupBy`, `Join`, `Max` y `Average`. Puede llamarlas directamente con la sintaxis de método en lugar de la sintaxis de consulta.

La sintaxis de consulta y la sintaxis de método son idénticas desde el punto de vista semántico, pero muchos usuarios creen que la sintaxis de consulta es mucho más sencilla y fácil de leer. Algunos métodos deben expresarse como llamadas de método. Por ejemplo, debe usar una llamada de método para expresar una consulta que recupera el número de elementos que cumplen una condición especificada. También debe usar una llamada de método para una consulta que recupera el elemento que tiene el valor máximo de una secuencia de origen. La documentación de referencia de los operadores de consulta estándar del espacio de nombres System.Linq generalmente usa la sintaxis de método. Por consiguiente, incluso cuando empiece a escribir consultas de LINK, resulta útil estar familiarizado con el uso de la sintaxis de método en las consultas y en las propias expresiones de consulta.

Métodos de extensión de operador de consulta estándar

En el ejemplo siguiente se muestra una *expresión de consulta* sencilla y la consulta equivalente desde el punto de vista semántico que se escribe como *consulta basada en métodos*.

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12};

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
    Output:
    6 8 10 12
    6 8 10 12
 */
```

El resultado de los dos ejemplos es idéntico. Como puede ver, el tipo de variable de consulta es el mismo en ambos formularios: IEnumerable<T>.

Para entender la consulta basada en métodos, vamos a examinarla más detenidamente. En el lado derecho de la expresión, observe que la cláusula `where` ahora se expresa como un método de instancia en el objeto `numbers`, que, como recordará, tiene un tipo de `IEnumerable<int>`. Si está familiarizado con la interfaz genérica IEnumerable<T>, sabe que no tiene un método `Where`. Pero si se invoca la lista de finalización de IntelliSense en el IDE de Visual Studio, verá no solo un método `Where`, sino muchos otros métodos tales como `Select`, `SelectMany`, `Join` y `Orderby`. Estos son todos los operadores de consulta estándar.

Aunque parece que IEnumerable<T> se haya redefinido para incluir estos métodos adicionales, en realidad no es el caso. Los operadores de consulta estándar se implementan como un nuevo tipo de método denominado *método de extensión*. Los métodos de extensión "extienden" un tipo existente; se pueden llamar como si fueran

métodos de instancia en el tipo. Los operadores de consulta estándar extienden IEnumerable<T> y esta es la razón por la que la puede escribir `numbers.Where(...)`.

Para empezar a usar LINK, lo único que realmente debe saber sobre los métodos de extensión es cómo incluirlos en el ámbito de la aplicación mediante el uso correcto de directivas `using`. Desde el punto de vista de la aplicación, un método de extensión y un método de instancia normal son iguales.

Para obtener más información sobre los métodos de extensión, vea Métodos de extensión. Para obtener más información sobre los operadores de consulta estándar, vea Información general sobre operadores de consulta estándar (C#). Algunos proveedores de LINQ, como LINQ to SQL y LINQ to XML, implementan sus propios operadores de consulta estándar y otros métodos de extensión además de IEnumerable<T>.

Expresiones lambda

En el ejemplo anterior, observe que la expresión condicional (`num % 2 == 0`) se pasa como argumento insertado al método `Where`: `Where(num => num % 2 == 0).` Esta expresión insertada se denomina expresión lambda. Se trata de una forma cómoda de escribir código que, de lo contrario, tendría que escribirse de forma más compleja como un método anónimo, un delegado genérico o un árbol de expresión. En C#, `=>` es el operador lambda, que se lee como "va a". La `num` situada a la izquierda del operador es la variable de entrada que corresponde a `num` en la expresión de consulta. El compilador puede deducir el tipo de `num` porque sabe que `numbers` es un tipo IEnumerable<T> genérico. El cuerpo de la expresión lambda es exactamente igual que la expresión de la sintaxis de consulta o de cualquier otra expresión o instrucción de C#; puede incluir llamadas de método y otra lógica compleja. El "valor devuelto" es simplemente el resultado de la expresión.

Para empezar a usar LINK, no es necesario emplear muchas expresiones lambda. Pero determinadas consultas solo se pueden expresar en sintaxis de método y algunas requieren expresiones lambda. Cuando esté más familiarizado con las expresiones lambda, verá que se trata de una herramienta eficaz y flexible del cuadro de herramientas de LINK. Para obtener más información, vea Expresiones lambda.

Capacidad de composición de consultas

En el ejemplo de código anterior, tenga en cuenta que el método `OrderBy` se invoca mediante el operador de punto en la llamada a `Where`. `Where` crea una secuencia filtrada y, luego, `Orderby` actúa en dicha secuencia ordenándola. Dado que las consultas devuelven un `IEnumerable`, redáctelas con la sintaxis de método encadenando las llamadas de método. Es lo que el compilador hace en segundo plano cuando se escriben consultas con la sintaxis de consulta. Y dado que una variable de consulta no almacena los resultados de la consulta, puede modificarla o usarla como base para una nueva consulta en cualquier momento, incluso después de que se haya ejecutado.

Características de C# compatibles con LINQ

29/06/2020 • 7 minutes to read • Edit Online

La siguiente sección presenta las nuevas construcciones de lenguaje incluidas en C# 3.0. Aunque estas nuevas características se usan hasta cierto punto con consultas LINQ, no se limitan a LINQ y se pueden usar en cualquier contexto en las que se consideren de utilidad.

Expresiones de consulta

Las expresiones de consulta usan una sintaxis declarativa similar a SQL o XQuery para consultar colecciones de IEnumerable. En tiempo de compilación, la sintaxis de consulta se convierte en llamadas de método a la implementación de un proveedor de LINQ de los métodos de extensión de operador de consulta estándar. Las aplicaciones controlan los operadores de consulta estándar que están en el ámbito al especificar el espacio de nombres adecuado con una directiva `using`. La siguiente expresión de consulta toma una matriz de cadenas, las agrupa por el primer carácter de la cadena y ordena los grupos.

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Para obtener más información, vea Expresiones de consulta LINQ.

Variables con asignación implícita de tipos (var)

En lugar de especificar explícitamente un tipo al declarar e inicializar una variable, se puede usar el modificador var para indicar al compilador que deduzca y asigne el tipo, como se muestra aquí:

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Las variables declaradas como `var` son tan fuertemente tipadas como las variables cuyo tipo se especifica explícitamente. El uso de `var` hace posible crear tipos anónimos, pero solo se puede usar para variables locales. También se pueden declarar matrices con asignación implícita de tipos.

Para más información, vea Variables locales con asignación implícita de tipos.

Inicializadores de objeto y colección

Los inicializadores de objeto y colección permiten inicializar objetos sin llamar explícitamente a un constructor para el objeto. Los inicializadores normalmente se usan en expresiones de consulta cuando proyectan los datos de origen en un nuevo tipo de datos. Suponiendo que hay una clase denominada `Customer` con las propiedades públicas `Name` y `Phone`, el inicializador de objeto se puede usar como en el código siguiente:

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Continuando con nuestra clase `Customer`, suponga que hay un origen de datos denominado `IncomingOrders` y que

para cada pedido con un `OrderSize` grande, nos gustaría crear un nuevo `Customer` basado en ese orden. Se pueden ejecutar una consulta LINQ en este origen de datos y usar la inicialización de objetos para rellenar una colección:

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone = o.Phone };
```

El origen de datos puede tener más propiedades ocultas en el montón que la clase `Customer`, como `OrderSize`, pero con la inicialización de objetos, los datos devueltos por la consulta se moldean en el tipo de datos deseado; elegimos los datos que son relevantes para nuestra clase. Como resultado, ahora tenemos un `IEnumerable` relleno con el nuevo `Customer` que queríamos. Lo anterior también se puede escribir en la sintaxis de método de LINQ:

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name =
y.Name, Phone = y.Phone });
```

Para obtener más información, consulte:

- Inicializadores de objeto y colección
- Sintaxis de las expresiones de consulta para operadores de consulta estándar

Tipos anónimos

Un tipo anónimo se construye por el compilador y el nombre del tipo solo está disponible para el compilador. Los tipos anónimos son una manera cómoda de agrupar un conjunto de propiedades temporalmente en un resultado de consulta sin tener que definir un tipo con nombre independiente. Los tipos anónimos se inicializan con una nueva expresión y un inicializador de objeto, como se muestra aquí:

```
select new {name = cust.Name, phone = cust.Phone};
```

Para obtener más información, vea Tipos anónimos.

Métodos de extensión.

Un método de extensión es un método estático que se puede asociar con un tipo, por lo que puede llamarse como si fuera un método de instancia en el tipo. Esta característica permite, en efecto, "agregar" nuevos métodos a los tipos existentes sin tener que modificarlos realmente. Los operadores de consulta estándar son un conjunto de métodos de extensión que proporcionan funciones de consultas LINQ para cualquier tipo que implemente IEnumerable<T>.

Para obtener más información, vea Métodos de extensión.

Expresiones lambda

Una expresión lambda es una función insertada que usa el operador => para separar los parámetros de entrada del cuerpo de la función y que se puede convertir en tiempo de compilación en un delegado o un árbol de expresión. En la programación de LINQ, encontrará expresiones lambda al realizar llamadas de método directas a los operadores de consulta estándar.

Para obtener más información, consulte:

- Funciones anónimas
- Expresiones lambda

- Árboles de expresión (C#)

Vea también

- Language Integrated Query (LINQ) (C#)

Tutorial: Escribir consultas en C# (LINQ)

18/03/2020 • 17 minutes to read • Edit Online

Este tutorial muestra las características del lenguaje C# que se usan para escribir expresiones de consulta de LINQ.

Crear un proyecto de C#

> **NOTE**
> Las siguientes instrucciones se aplican a Visual Studio. Si usa otro entorno de desarrollo, cree un proyecto de consola con una referencia a System.Core.dll y una directiva `using` para el espacio de nombres System.Linq.

Para crear un proyecto en Visual Studio

1. Inicie Visual Studio.
2. En la barra de menús, elija **Archivo**, **Nuevo**, **Proyecto**.

   Aparece el cuadro de diálogo **Nuevo proyecto** .
3. Expanda **Instalado**, **Plantillas**, **Visual C#** y luego elija **Aplicación de consola**.
4. En el cuadro de texto **Nombre**, escriba otro nombre o acepte el predeterminado y luego elija el botón **Aceptar**.

   El nuevo proyecto aparece en el **Explorador de soluciones**.
5. Observe que el proyecto tiene una referencia a System.Core.dll y una directiva `using` para el espacio de nombres System.Linq.

Crear un origen de datos en memoria

El origen de datos de las consultas es una simple lista de objetos `Student`. Cada registro `Student` tiene un nombre, un apellido y una matriz de enteros que representa sus puntuaciones de las pruebas en la clase. Copie este código en el proyecto. Observe las siguientes características:

- La clase `Student` consta de propiedades implementadas automáticamente.
- Cada alumno de la lista se inicializa con un inicializador de objeto.
- La propia lista se inicializa con un inicializador de colección.

Toda la estructura de datos se inicializará y creará una instancia sin llamadas explícitas a ningún constructor ni acceso a miembro explícito. Para obtener más información sobre estas nuevas características, vea Propiedades autoimplementadas (Propiedades implementadas automáticamente) y Inicializadores de objeto y de colección.

Para agregar el origen de datos

- Agregue la clase `Student` y la lista inicializada de alumnos a la clase `Program` del proyecto.

```
public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91}}
};
```

Para agregar un nuevo alumno a la lista de alumnos

1. Agregue un nuevo `Student` a la lista `Students` y use el nombre y las puntuaciones de las pruebas que prefiera. Pruebe a escribir toda la nueva información de alumno para conocer mejor la sintaxis del inicializador de objeto.

Crear la consulta

Para crear una consulta simple

- En el método `Main` de la aplicación, cree una consulta simple que, cuando se ejecute, genere una lista de todos los alumnos cuya puntuación en la primera prueba haya sido superior a 90. Tenga en cuenta que, dado que se ha seleccionado todo el objeto `Student`, el tipo de la consulta es `IEnumerable<Student>`. Aunque el código podría usar tipos implícitos con la palabra clave var, se usan tipos explícitos para mostrar claramente los resultados. (Para obtener más información sobre `var`, vea Implicitly Typed Local Variables [Variables locales con tipo implícito]).

  Tenga también en cuenta que la variable de rango de la consulta, `student`, actúa como referencia a cada `Student` del origen, lo que proporciona acceso a miembro para cada objeto.

```
// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

Ejecutar la consulta

Para ejecutar la consulta

1. Escriba ahora el bucle `foreach` que hará que la consulta se ejecute. Tenga en cuenta los siguiente sobre el código:

   - A cada elemento de la secuencia devuelta se accede mediante la variable de iteración del bucle `foreach`.

- El tipo de esta variables es `Student` y el tipo de la variable de consulta es compatible, `IEnumerable<Student>`.

2. Tras agregar este código, compile y ejecute la aplicación para ver los resultados en la ventana **Consola**.

```
// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
{
    Console.WriteLine("{0}, {1}", student.Last, student.First);
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

Para agregar otra condición de filtro

1. Puede combinar varias condiciones booleanas en la cláusula `where` para delimitar aún más una consulta. El código siguiente agrega una condición para que la consulta devuelva los alumnos cuya primera puntuación haya sido superior a 90 y cuya última puntuación haya sido inferior a 80. La cláusula `where` debería ser similar al código siguiente.

   ```
   where student.Scores[0] > 90 && student.Scores[3] < 80
   ```

   Para obtener más información, vea where (Cláusula).

Modificar la consulta

Para ordenar los resultados

1. Le resultará más fácil examinar los resultados si se muestran con algún tipo de orden. Puede ordenar la secuencia devuelta por cualquier campo accesible de los elementos de origen. Por ejemplo, la cláusula `orderby` siguiente ordena los resultados alfabéticamente de la A a la Z por el apellido de cada alumno. Agregue la cláusula `orderby` siguiente a la consulta, inmediatamente después de la instrucción `where` y antes de la instrucción `select`:

   ```
   orderby student.Last ascending
   ```

2. Cambie ahora la cláusula `orderby` para que ordene los resultados en orden inverso según la puntuación en la primera prueba, de la puntuación más alta a la más baja.

   ```
   orderby student.Scores[0] descending
   ```

3. Cambie la cadena de formato `WriteLine` para poder ver las puntuaciones:

   ```
   Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
   ```

   Para obtener más información, vea orderby (Cláusula).

Para agrupar los resultados

1. La agrupación es una funcionalidad de gran eficacia en expresiones de consulta. Una consulta con una cláusula group genera una secuencia de grupos y cada grupo propiamente dicho contiene un `Key` y una secuencia que consta de todos los miembros del grupo. La siguiente nueva consulta agrupa los alumnos usando la primera letra del apellido como clave.

```
// studentQuery2 is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

2. Tenga en cuenta que el tipo de la consulta ha cambiado. Ahora genera una secuencia de grupos que tienen un tipo `char` como clave y una secuencia de objetos `Student`. Dado que el tipo de la consulta ha cambiado, el siguiente código cambia también el bucle de ejecución `foreach`:

```
// studentGroup is a IGrouping<char, Student>
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine("   {0}, {1}",
                  student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

3. Ejecute la aplicación y vea los resultados en la ventana **Consola**.

   Para obtener más información, vea group (Cláusula).

Para que las variables tengan tipo implícito

1. La codificación explícita con `IEnumerables` de `IGroupings` puede resultar tediosa. Puede escribir la misma consulta y el bucle `foreach` mucho más cómodamente con `var`. La palabra clave `var` no cambia los tipos de los objetos; solo indica al compilador que deduzca los tipos. Cambie el tipo de `studentQuery` y la variable de iteración `group` a `var` y vuelva a ejecutar la consulta. Tenga en cuenta que en el bucle `foreach` interior, la variable de iteración sigue teniendo como tipo `Student` y la consulta funciona igual que antes. Cambie la variable de iteración `s` a `var` y vuelva a ejecutar la consulta. Como puede ver, obtiene exactamente los mismos resultados.

```
var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Para obtener más información sobre var, vea Variables locales con asignación implícita de tipos.

Para ordenar los grupos por su valor clave

1. Al ejecutar la consulta anterior, observará que los grupos no están en orden alfabético. Para cambiar esto, debe especificar una cláusula `orderby` después de la cláusula `group`. Pero para usar una cláusula `orderby`, necesita primero un identificador que actúe como referencia a los grupos creados por la cláusula `group`. El identificador se especifica con la palabra clave `into`, de la manera siguiente:

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("   {0}, {1}",
            student.Last, student.First);
    }
}

// Output:
//A
//   Adams, Terry
//F
//   Fakhouri, Fadi
//   Feng, Hanying
//G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
//M
//   Mortensen, Sven
//O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
//T
//   Tucker, Lance
//   Tucker, Michael
//Z
//   Zabokritski, Eugene
```

Cuando ejecute esta consulta, verá que los grupos aparecen ahora ordenados alfabéticamente.

Para incluir un identificador mediante let

1. Puede usar la palabra clave `let` para incluir un identificador con cualquier resultado de la expresión en la expresión de consulta. Este identificador puede resultar cómodo, como en el ejemplo siguiente, o mejorar el rendimiento almacenando los resultados de una expresión para que no tenga que calcularse varias veces.

```
// studentQuery5 is an IEnumerable<string>
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select student.Last + " " + student.First;

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
// Omelchenko Svetlana
// O'Donnell Claire
// Mortensen Sven
// Garcia Cesar
// Fakhouri Fadi
// Feng Hanying
// Garcia Hugo
// Adams Terry
// Zabokritski Eugene
// Tucker Michael
```

Para obtener más información, vea let (Cláusula).

Para usar la sintaxis de método en una expresión de consulta

1. Tal y como se describe en Sintaxis de consulta y sintaxis de método en LINQ, algunas operaciones de consulta solo pueden expresarse con una sintaxis de método. El código siguiente calcula la puntuación total de cada `Student` de la secuencia de origen y luego llama al método `Average()` en los resultados de esa consulta para calcular la puntuación media de la clase.

```
var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

Para transformar o proyectar en la cláusula select

1. Es muy frecuente que una consulta genere una secuencia cuyos elementos difieren de los elementos de las secuencias de origen. Elimine la consulta y el bucle de ejecución anteriores o conviértalos en comentario, y reemplácelos por el código siguiente. Tenga en cuenta que la consulta devuelve una secuencia de cadenas (no `Students`) y este hecho se refleja en el bucle `foreach`.

```
IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

2. El código anterior en este tutorial indicaba que la puntuación media de la clase es aproximadamente 334. Para generar una secuencia de `Students` cuya puntuación total sea superior al promedio de la clase, junto con su `Student ID`, puede usar un tipo anónimo en la instrucción `select`:

```
var studentQuery8 =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in studentQuery8)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

Pasos siguientes

Cuando se haya familiarizado con los aspectos básicos del uso de consultas en C#, estará preparado para leer la documentación y ejemplos del tipo específico de proveedor LINQ que le interese:

LINQ to SQL

LINQ to DataSet

LINQ to XML (C#)

LINQ to Objects (C#)

Vea también

- Language Integrated Query (LINQ) (C#)
- Expresiones de consulta LINQ

Información general sobre operadores de consulta estándar (C#)

29/06/2020 • 6 minutes to read • Edit Online

Los *operadores de consulta estándar* son los métodos que constituyen el modelo LINQ. La mayoría de estos métodos funciona en secuencias; donde una secuencia es un objeto cuyo tipo implementa la interfaz IEnumerable<T> o la interfaz IQueryable<T>. Los operadores de consulta estándar ofrecen funcionalidades de consulta, como las funciones de filtrado, proyección, agregación y ordenación, entre otras.

Hay dos conjuntos de operadores de consulta estándar de LINQ: uno que actúa sobre objetos de tipo IEnumerable<T> y otro sobre objetos de tipo IQueryable<T>. Los métodos que forman cada conjunto son miembros estáticos de las clases Enumerable y Queryable, respectivamente. Se definen como *métodos de extensión* del tipo en el que actúan. Se puede llamar a los métodos de extensión mediante sintaxis de método estático o sintaxis de método de instancia.

Además, varios métodos de operador de consulta estándar funcionan en tipos distintos de los que se basan en IEnumerable<T> o IQueryable<T>. El tipo Enumerable define dos métodos que funcionan en objetos de tipo IEnumerable. Estos métodos, Cast<TResult>(IEnumerable) y OfType<TResult>(IEnumerable), le permiten habilitar una colección no genérica o no parametrizada para consultarse en el patrón LINQ. Para ello, se crea una colección de objetos fuertemente tipados. La clase Queryable define dos métodos similares, Cast<TResult>(IQueryable) y OfType<TResult>(IQueryable), que funcionan en objetos de tipo Queryable.

Los operadores de consulta estándar difieren en sus intervalos de ejecución, dependiendo de que devuelvan un valor singleton o una secuencia de valores. Esos métodos que devuelven un valor singleton (por ejemplo, Average y Sum) se ejecutan inmediatamente. Los métodos que devuelven una secuencia aplazan la ejecución de la consulta y devuelven un objeto enumerable.

En el caso de los métodos que actúan en colecciones en memoria, es decir, los métodos que extienden IEnumerable<T>, el objeto enumerable devuelto captura los argumentos que se han pasado al método. Cuando se enumera ese objeto, se emplea la lógica del operador de consulta y se devuelven los resultados de la consulta.

En cambio, los métodos que extienden IQueryable<T> no implementan ningún comportamiento de realización de consultas. Compilan un árbol de expresión que representa la consulta que se va a realizar. El procesamiento de consultas se controla mediante el objeto IQueryable<T> de origen.

Las llamadas a métodos de consulta se pueden encadenar juntas en una consulta, lo que permite que las consultas se conviertan en complejas de forma arbitraria.

En el ejemplo de código siguiente se muestra el uso de los operadores de consulta estándar para obtener información sobre una secuencia.

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Sintaxis de expresiones de consulta

Algunos de los operadores de consulta estándar que se usan con más frecuencia tienen una sintaxis especial de palabras clave dedicadas de lenguaje C# y Visual Basic para que se puedan invocar como parte de una *expresión de consulta*. Para obtener más información sobre los operadores de consulta estándar que incluyen palabras clave dedicadas y sus sintaxis correspondientes, vea Sintaxis de las expresiones de consulta para operadores de consulta estándar (C#).

Extender los operadores de consulta estándar

Puede aumentar el conjunto de operadores de consulta estándar creando métodos específicos de dominio que sean adecuados para su tecnología o dominio de destino. También puede reemplazar los operadores de consulta estándar con sus propias implementaciones que proporcionen otros servicios tales como evaluación remota, traducción de consultas y optimización. Vea AsEnumerable para obtener un ejemplo.

Secciones relacionadas

Los vínculos siguientes llevan a artículos que ofrecen información adicional sobre los distintos operadores de consulta estándar según la funcionalidad.

Ordenación de datos [C#]

Operaciones set [C#]

Filtrado de datos [C#]

Operaciones cuantificadoras [C#]

Operaciones de proyección [C#]

Realizar particiones de datos [C#]

Operaciones de combinación [C#]

Agrupar datos [C#]

Operaciones de generación [C#]

Operaciones de igualdad [C#]

Operaciones de elementos [C#]

Convertir tipos de datos [C#]

Operaciones de concatenación [C#]

Operaciones de agregación [C#]

Vea también

- Enumerable
- Queryable
- Introducción a las consultas LINQ (C#)
- Sintaxis de las expresiones de consulta para operadores de consulta estándar (C#)
- Clasificación de operadores de consulta estándar por modo de ejecución (C#)
- Métodos de extensión

Sintaxis de las expresiones de consulta para operadores de consulta estándar (C#)

18/03/2020 • 2 minutes to read • Edit Online

Algunos de los operadores de consulta estándar que se usan con más frecuencia tienen una sintaxis especial de palabras clave de lenguaje C# para que se puedan invocar como parte de una *expresión de consulta*. Una expresión de consulta constituye una forma diferente de expresar una consulta, más legible que su equivalente *basada en métodos*. Las cláusulas de las expresiones de consulta se convierten en llamadas a los métodos de consulta en tiempo de compilación.

Tabla de sintaxis de expresiones de consulta

En la tabla siguiente se muestran los operadores de consulta estándar que poseen cláusulas de expresiones de consulta equivalentes.

MÉTODO	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#
Cast	Use una variable de rango con tipo explícito, por ejemplo: `from int i in numbers` (Para obtener más información, vea Cláusula from).
GroupBy	`group … by` o bien `group … by … into …` (Para obtener más información, vea Cláusula group).
GroupJoin<TOuter,TInner,TKey,TResult> (IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>,TResult>)	`join … in … on … equals … into …` (Para obtener más información, vea join (Cláusula, Referencia de C#)).
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (Para obtener más información, vea join (Cláusula, Referencia de C#)).
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (Para obtener más información, vea orderby (Cláusula)).
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (Para obtener más información, vea orderby (Cláusula)).

MÉTODO	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#
Select	`select` (Para obtener más información, vea Cláusula select).
SelectMany	Varias cláusulas `from`. (Para obtener más información, vea Cláusula from).
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (Para obtener más información, vea orderby (Cláusula)).
ThenByDescending<TSource,TKey> (IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (Para obtener más información, vea orderby (Cláusula)).
Where	`where` (Para obtener más información, vea where (Cláusula)).

Vea también

- Enumerable
- Queryable
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Clasificación de operadores de consulta estándar por modo de ejecución (C#)

Clasificación de operadores de consulta estándar por modo de ejecución (C#)

18/03/2020 • 6 minutes to read • Edit Online

Las implementaciones de LINQ to Objects de los métodos de operador de consulta estándar se ejecutan de una de dos formas principales: inmediata o aplazada. Los operadores de consulta que usan la ejecución aplazada pueden dividirse además en dos categorías: de streaming y de no streaming. Si sabe cómo se ejecutan los diferentes operadores de consulta, puede servirle para entender los resultados que se obtienen de una consulta determinada. Esto es especialmente cierto si se está cambiando el origen de datos o si se está creando una consulta sobre otra. En este tema se clasifican los operadores de consulta estándar según su modo de ejecución.

Modos de ejecución

Inmediato

La ejecución inmediata significa que se lee el origen de datos y la operación se realiza en el punto en el código donde se declara la consulta. Todos los operadores de consulta estándar que devuelven un resultado único no enumerable se ejecutan de manera inmediata.

Aplazada

La ejecución aplazada significa que la operación no se realiza en el punto en el código donde se declara la consulta. La operación se realiza solo cuando se enumera la variable de consulta, por ejemplo, mediante una instrucción `foreach`. Esto significa que los resultados de ejecutar la consulta dependen del contenido del origen de datos cuando se ejecuta la consulta en lugar de cuando se define la consulta. Si la variable de consulta se enumera varias veces, es posible que los resultados difieran cada vez. Casi todos los operadores de consulta estándar cuyo tipo de valor devuelto es IEnumerable<T> o IOrderedEnumerable<TElement> se ejecutan de una manera diferida.

Los operadores de consulta que usan la ejecución aplazada pueden clasificarse además como de streaming o de no streaming.

Streaming

Los operadores de streaming no deben leer todos los datos de origen antes de que produzcan elementos. En el momento de la ejecución, un operador de streaming realiza su operación en cada elemento de origen mientras se lee y proporciona el elemento si es necesario. Un operador de streaming continúa leyendo los elementos de origen hasta que se puede generar un elemento de resultado. Esto significa que es posible leer más de un elemento de origen para generar un elemento de resultado.

No streaming

Los operadores de no streaming deben leer todos los datos de origen antes de poder proporcionar un elemento de resultado. Las operaciones como la ordenación o la agrupación pertenecen a esta categoría. En tiempo de ejecución, los operadores de consulta de no streaming leen todos los datos de origen, los colocan en una estructura de datos, realizan la operación y proporcionan los elementos resultantes.

Tabla de clasificación

En la tabla siguiente se clasifica cada método de operador de consulta estándar según su método de ejecución.

NOTE

Si un operador se marca en dos columnas, dos secuencias de entrada intervienen en la operación, y cada secuencia se evalúa de manera diferente. En estos casos, siempre es la primera secuencia de la lista de parámetros la que se evalúa en un modo de transmisión diferido.

OPERADOR DE CONSULTA ESTÁNDAR	TIPO DEVUELTO	EJECUCIÓN INMEDIATA	EJECUCIÓN APLAZADA DE STREAMING	EJECUCIÓN APLAZADA DE NO STREAMING
Aggregate	TSource	x		
All	Boolean	x		
Any	Boolean	x		
AsEnumerable	IEnumerable<T>		x	
Average	Valor numérico único	x		
Cast	IEnumerable<T>		x	
Concat	IEnumerable<T>		x	
Contains	Boolean	x		
Count	Int32	x		
DefaultIfEmpty	IEnumerable<T>		x	
Distinct	IEnumerable<T>		x	
ElementAt	TSource	x		
ElementAtOrDefault	TSource	x		
Empty	IEnumerable<T>	x		
Except	IEnumerable<T>		x	x
First	TSource	x		
FirstOrDefault	TSource	x		
GroupBy	IEnumerable<T>			x
GroupJoin	IEnumerable<T>		x	x
Intersect	IEnumerable<T>		x	x
Join	IEnumerable<T>		x	x

OPERADOR DE CONSULTA ESTÁNDAR	TIPO DEVUELTO	EJECUCIÓN INMEDIATA	EJECUCIÓN APLAZADA DE STREAMING	EJECUCIÓN APLAZADA DE NO STREAMING
Last	TSource	x		
LastOrDefault	TSource	x		
LongCount	Int64	x		
Max	Valor numérico único, TSource o TResult	x		
Min	Valor numérico único, TSource o TResult	x		
OfType	IEnumerable<T>		x	
OrderBy	IOrderedEnumerable<TElement>			x
OrderByDescending	IOrderedEnumerable<TElement>			x
Range	IEnumerable<T>		x	
Repeat	IEnumerable<T>		x	
Reverse	IEnumerable<T>			x
Select	IEnumerable<T>		x	
SelectMany	IEnumerable<T>		x	
SequenceEqual	Boolean	x		
Single	TSource	x		
SingleOrDefault	TSource	x		
Skip	IEnumerable<T>		x	
SkipWhile	IEnumerable<T>		x	
Sum	Valor numérico único	x		
Take	IEnumerable<T>		x	
TakeWhile	IEnumerable<T>		x	
ThenBy	IOrderedEnumerable<TElement>			x

OPERADOR DE CONSULTA ESTÁNDAR	TIPO DEVUELTO	EJECUCIÓN INMEDIATA	EJECUCIÓN APLAZADA DE STREAMING	EJECUCIÓN APLAZADA DE NO STREAMING
ThenByDescending	IOrderedEnumerable<TElement>			x
ToArray	Matriz de TSource	x		
ToDictionary	Dictionary<TKey,TValue>	x		
ToList	IList<T>	x		
ToLookup	ILookup<TKey,TElement>	x		
Union	IEnumerable<T>		x	
Where	IEnumerable<T>		x	

Vea también

- Enumerable
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Sintaxis de las expresiones de consulta para operadores de consulta estándar (C#)
- LINQ to Objects (C#)

Ordenación de datos (C#)

18/03/2020 • 4 minutes to read • Edit Online

Una operación de ordenación ordena los elementos de una secuencia según uno o varios atributos. El primer criterio de ordenación realiza una ordenación primaria de los elementos. Al especificar un segundo criterio de ordenación, se pueden ordenar los elementos dentro de cada grupo de ordenación primaria.

En la ilustración siguiente se muestran los resultados de una operación de ordenación alfabética en una secuencia de caracteres:

Los métodos de operador de consulta estándar que ordenan datos de datos se enumeran en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
OrderBy	Ordena valores en orden ascendente.	`orderby`	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	Ordena valores en orden descendente.	`orderby … descending`	Enumerable.OrderByDescending Queryable.OrderByDescending
ThenBy	Realiza una ordenación secundaria en orden ascendente.	`orderby …, …`	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Realiza una ordenación secundaria en orden descendente.	`orderby …, … descending`	Enumerable.ThenByDescending Queryable.ThenByDescending
Reverse	Invierte el orden de los elementos de una colección.	No disponible.	Enumerable.Reverse Queryable.Reverse

Ejemplos de sintaxis de expresiones de consulta

Ejemplos de ordenación principal

Orden ascendente principal

En el siguiente ejemplo se muestra cómo usar la cláusula `orderby` en una consulta LINQ para ordenar las cadenas de una matriz por la longitud de la cadena, en orden ascendente.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    brown
    jumps
*/
```

Orden descendente principal

En el siguiente ejemplo se muestra cómo usar la cláusula `orderby descending` en una consulta LINQ para ordenar las cadenas por su letra inicial, en orden descendente.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    quick
    jumps
    fox
    brown
*/
```

Ejemplos de ordenación secundaria

Orden ascendente secundario

En el siguiente ejemplo se muestra cómo usar la cláusula `orderby` en una consulta LINQ para realizar una ordenación principal y secundaria de las cadenas de una matriz. Las cadenas se ordenan primero por su longitud y, después, por la letra inicial de la cadena, en orden ascendente.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1)
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    fox
    the
    brown
    jumps
    quick
*/
```

Orden descendente secundario

En el siguiente ejemplo se muestra cómo usar la cláusula `orderby descending` en una consulta LINQ para realizar una ordenación principal en orden ascendente y una ordenación secundaria en orden descendente. Las cadenas se ordenan primero por su longitud y, después, por la letra inicial de la cadena.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            orderby word.Length, word.Substring(0, 1) descending
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
    quick
    jumps
    brown
*/
```

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- orderby (cláusula)
- Ordenar los resultados de una cláusula join
- Procedimiento para ordenar o filtrar datos de texto por palabra o campo (LINQ) (C#)

Operaciones set (C#)

18/03/2020 • 3 minutes to read • Edit Online

Las operaciones set de LINQ se refieren a operaciones de consulta que generan un conjunto de resultados en función de la presencia o ausencia de elementos equivalentes dentro de la misma colección o en distintas colecciones (o conjuntos).

Los métodos del operador de consulta estándar que realizan operaciones set se indican en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Distinct	Quita valores duplicados de una colección.	No disponible.	Enumerable.Distinct Queryable.Distinct
Except	Devuelve la diferencia de conjuntos, es decir, los elementos de una colección que no aparecen en una segunda colección.	No disponible.	Enumerable.Except Queryable.Except
Formar intersección	Devuelve la intersección de conjuntos, es decir, los elementos que aparecen en las dos colecciones.	No disponible.	Enumerable.Intersect Queryable.Intersect
Union	Devuelve la unión de conjuntos, es decir, los elementos únicos que aparecen en una de las dos colecciones.	No disponible.	Enumerable.Union Queryable.Union

Comparación de operaciones set

Distinct

En la siguiente ilustración se muestra el comportamiento del método Enumerable.Distinct en una secuencia de caracteres. La secuencia devuelta contiene los elementos únicos de la secuencia de entrada.

```
string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth" };

IEnumerable<string> query = from planet in planets.Distinct()
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Mars
 */
```

Except

En el ejemplo siguiente se muestra el comportamiento de Enumerable.Except. La secuencia devuelta solo contiene los elementos de la primera secuencia de entrada que no están en la segunda secuencia de entrada.

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Except(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Venus
 */
```

Formar intersección

En el ejemplo siguiente se muestra el comportamiento de Enumerable.Intersect. La secuencia devuelta contiene los elementos que son comunes a las dos secuencias de entrada.

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Intersect(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */
```

Union

En el siguiente ejemplo se muestra una operación de unión en dos secuencias de caracteres. La secuencia devuelta contiene los elementos únicos de las dos secuencias de entrada.

```
string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Union(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Jupiter
 * Mars
 */
```

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Procedimiento para combinar y comparar colecciones de cadenas (LINQ) (C#)
- Procedimiento para buscar la diferencia de conjuntos entre dos listas (LINQ) (C#)

Filtrado de datos (C#)

18/03/2020 • 2 minutes to read • Edit Online

El filtrado hace referencia a la operación de restringir el conjunto de resultados, de manera que solo contenga los elementos que cumplen una condición especificada. También se conoce como selección.

En la ilustración siguiente se muestran los resultados de filtrar una secuencia de caracteres. El predicado de la operación de filtrado especifica que el carácter debe ser "A".

Los métodos del operador de consulta estándar que realizan selecciones se indican en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
OfType	Selecciona valores en función de su capacidad para convertirse en un tipo especificado.	No disponible.	Enumerable.OfType Queryable.OfType
Where	Selecciona valores basados en una función de predicado.	`where`	Enumerable.Where Queryable.Where

Ejemplo de sintaxis de expresiones de consulta

En el siguiente ejemplo se usa la cláusula `where` para filtrar de una matriz aquellas cadenas que tienen una longitud específica.

```
string[] words = { "the", "quick", "brown", "fox", "jumps" };

IEnumerable<string> query = from word in words
                            where word.Length == 3
                            select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

    the
    fox
*/
```

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- where (cláusula)
- Especificar dinámicamente filtros con predicado en tiempo de ejecución
- Procedimiento para consultar los metadatos de un ensamblado con reflexión (LINQ) (C#)
- Procedimiento para buscar archivos con un nombre o atributo especificados (C#)
- Procedimiento para ordenar o filtrar datos de texto por palabra o campo (LINQ) (C#)

Operaciones cuantificadoras (C#)

18/03/2020 • 4 minutes to read • Edit Online

Las operaciones cuantificadoras devuelven un valor Boolean que indica si algunos o todos los elementos de una secuencia cumplen una condición.

En la siguiente ilustración se muestran dos operaciones cuantificadoras diferentes en dos secuencias de origen distintas. La primera operación pregunta si uno o varios de los elementos son el carácter "A", y el resultado es `true`. La segunda operación pregunta si todos los elementos son el carácter "A", y el resultado es `true`.

Los métodos del operador de consulta estándar que realizan operaciones cuantificadoras se indican en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Todas	Determina si todos los elementos de una secuencia cumplen una condición.	No disponible.	Enumerable.All Queryable.All
Cualquiera	Determina si algunos de los elementos de una secuencia cumplen una condición.	No disponible.	Enumerable.Any Queryable.Any
Contiene	Determina si una secuencia contiene un elemento especificado.	No disponible.	Enumerable.Contains Queryable.Contains

Ejemplos de sintaxis de expresiones de consulta

Todas

En el ejemplo siguiente se usa `All` para comprobar que todas las cadenas tienen una longitud específica.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                                where market.Items.All(item => item.Length == 5)
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}
```

Cualquiera

En el ejemplo siguiente se usa `Any` para comprobar que las cadenas se inician con "o".

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have any fruit names start with 'o'
    IEnumerable<string> names = from market in markets
                                where market.Items.Any(item => item.StartsWith("o"))
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
    // Adam's market
}
```

Contiene

En el ejemplo siguiente se usa `Contains` para comprobar que una matriz tenga un elemento específico.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market contains fruit names equal 'kiwi'
    IEnumerable<string> names = from market in markets
                                where market.Items.Contains("kiwi")
                                select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Emily's market
    // Adam's market
}
```

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Especificar dinámicamente filtros con predicado en tiempo de ejecución
- Procedimiento para buscar frases que contengan un conjunto especificado de palabras (LINQ) (C#)

Operaciones de proyección (C#)

18/03/2020 • 6 minutes to read • Edit Online

El término "proyección" hace referencia a la operación de transformar un objeto en una nueva forma que, a menudo, consta solo de aquellas propiedades que se usarán posteriormente. Utilizando la proyección, puede construir un tipo nuevo creado a partir de cada objeto. Se puede proyectar una propiedad y realizar una función matemática en ella. También puede proyectar el objeto original sin cambiarlo.

Los métodos del operador de consulta estándar que realizan proyecciones se indican en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Seleccionar	Proyecta valores basados en una función de transformación.	`select`	Enumerable.Select Queryable.Select
SelectMany	Proyecta secuencias de valores que se basan en una función de transformación y después los convierte en una secuencia.	Use varias cláusulas `from`	Enumerable.SelectMany Queryable.SelectMany

Ejemplos de sintaxis de expresiones de consulta

Seleccionar

En el ejemplo siguiente se usa la cláusula `select` para proyectar la primera letra de cada cadena de una lista de cadenas.

```
List<string> words = new List<string>() { "an", "apple", "a", "day" };

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    a
    a
    a
    d
*/
```

SelectMany

En el ejemplo siguiente se usan varias cláusulas `from` para proyectar cada palabra de todas las cadenas de una lista de cadenas.

```
List<string> phrases = new List<string>() { "an apple a day", "the quick brown fox" };

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
    Console.WriteLine(s);

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

Select frente a SelectMany

La función tanto de `Select()` como de `SelectMany()` consiste en generar un valor (o valores) de resultado a partir de valores de origen. `Select()` genera un valor de resultado para cada valor de origen. El resultado global, por tanto, es una colección que tiene el mismo número de elementos que la colección de origen. En cambio, `SelectMany()` genera un resultado global único que contiene subcolecciones concatenadas procedentes de cada valor de origen. La función de transformación que se pasa como argumento a `SelectMany()` debe devolver una secuencia enumerable de valores para cada valor de origen. Luego, `SelectMany()` concatena estas secuencias enumerables para crear una secuencia de gran tamaño.

Las dos ilustraciones siguientes muestran la diferencia conceptual entre las acciones de estos dos métodos. En cada caso, se supone que la función de selector (transformación) selecciona la matriz de flores de cada valor de origen.

En esta ilustración se muestra cómo `Select()` devuelve una colección que tiene el mismo número de elementos que la colección de origen.

En esta ilustración se muestra cómo `SelectMany()` concatena la secuencia intermedia de matrices en un valor de resultado final que contiene cada uno de los valores de todas las matrices intermedias.

Ejemplo de código

En el ejemplo siguiente se compara el comportamiento de `Select()` y `SelectMany()`. El código crea un "ramo" de flores tomando los dos primeros elementos de cada lista de nombres de flores de la colección de origen. En este ejemplo, el "valor único" que la función de transformación Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>) usa es una colección de valores. Para ello, se requiere el bucle adicional `foreach` a fin de enumerar cada una de las cadenas de cada subsecuencia.

```
class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new List<Bouquet>() {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "larkspur" }},
        new Bouquet{ Flowers = new List<string> { "tulip", "rose", "orchid" }},
        new Bouquet{ Flowers = new List<string> { "gladiolis", "lily", "snapdragon", "aster", "protea" }},
        new Bouquet{ Flowers = new List<string> { "larkspur", "lilac", "iris", "dahlia" }}
    };

    // *********** Select ***********
    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    // ********* SelectMany *********
    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<String> collection in query1)
        foreach (string item in collection)
            Console.WriteLine(item);

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
        Console.WriteLine(item);

    /* This code produces the following output:

       Results by using Select():
        sunflower
        daisy
        daffodil
        larkspur
        tulip
        rose
        orchid
        gladiolis
        lily
        snapdragon
```

```
        snapdragon
        aster
        protea
        larkspur
        lilac
        iris
        dahlia

       Results by using SelectMany():
        sunflower
        daisy
        daffodil
        larkspur
        tulip
        rose
        orchid
        gladiolis
        lily
        snapdragon
        aster
        protea
        larkspur
        lilac
        iris
        dahlia
    */

}
```

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- select (cláusula)
- Procedimiento para rellenar colecciones de objetos de varios orígenes (LINQ) (C#)
- Procedimiento para dividir un archivo en varios mediante el uso de grupos (LINQ) (C#)

Realizar particiones de datos (C#)

18/03/2020 • 2 minutes to read • Edit Online

Partición en LINQ es la operación de dividir una secuencia de entrada en dos secciones, sin reorganizar los elementos, y devolver después una de las secciones.

En la siguiente ilustración se muestran los resultados de tres operaciones de partición diferentes en una secuencia de caracteres. La primera operación devuelve los tres primeros elementos de la secuencia. La segunda operación omite los tres primeros elementos y devuelve los restantes. La tercera operación omite los dos primeros elementos de la secuencia y devuelve los tres siguientes.

Los métodos de operador de consulta estándar que realizan particiones de las secuencias se enumeran en la sección siguiente.

Operadores

NOMBRE DEL OPERADOR	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Skip	Omite los elementos hasta una determinada posición de una secuencia.	No disponible.	Enumerable.Skip Queryable.Skip
SkipWhile	Omite los elementos según una función de predicado hasta que un elemento no satisface la condición.	No disponible.	Enumerable.SkipWhile Queryable.SkipWhile
Take	Admite los elementos hasta una determinada posición de una secuencia.	No disponible.	Enumerable.Take Queryable.Take
TakeWhile	Admite los elementos según una función de predicado hasta que un elemento no satisface la condición.	No disponible.	Enumerable.TakeWhile Queryable.TakeWhile

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))

Operaciones de combinación (C#)

18/03/2020 • 6 minutes to read • Edit Online

Una *combinación* de dos orígenes de datos es la asociación de objetos de un origen de datos con los objetos que comparten un atributo común en otro origen de datos.

La combinación es una operación importante en las consultas destinadas a orígenes de datos cuyas relaciones entre sí no se puede seguir directamente. En la programación orientada a objetos, esto podría significar una correlación entre objetos que no está modelada, como el sentido contrario de una relación unidireccional. Un ejemplo de una relación unidireccional es una clase Cliente que tiene una propiedad de tipo Ciudad, pero la clase Ciudad no tiene una propiedad que sea una colección de objetos Cliente. Si tiene una lista de objetos Ciudad y quiere encontrar todos los clientes en cada ciudad, podría usar una operación de combinación para encontrarlos.

Los métodos de combinación que se han proporcionado en el marco de LINQ son Join y GroupJoin. Estos métodos efectúan combinaciones de igualdad, o combinaciones que hacen corresponder dos orígenes de datos en función de la igualdad de sus claves. (Para comparar, Transact-SQL admite otros operadores de combinación aparte de 'igual', por ejemplo, 'menor que'). En términos de base de datos relacional, Join implementa una combinación interna, un tipo de combinación en la que solo se devuelven los objetos que tienen una correspondencia en el otro conjunto de datos. El método GroupJoin no tiene equivalente directo en términos de bases de datos relacionales; pero implementa un superconjunto de combinaciones internas y combinaciones externas izquierdas. Una combinación externa izquierda es una combinación que devuelve cada elemento del primer origen de datos (izquierda), aunque no tenga elementos correlacionados en el otro origen de datos.

En la ilustración siguiente se muestra una vista conceptual de dos conjuntos y los elementos de esos conjuntos que se incluyen en una combinación interna o en una combinación externa izquierda.

Métodos

NOMBRE DEL MÉTODO	DESCRIPCIÓN	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Join	Combina dos secuencias según las funciones de selector de claves y extrae pares de valores.	`join … in … on … equals …`	Enumerable.Join Queryable.Join
GroupJoin	Combina dos secuencias según las funciones de selector de claves y agrupa los resultados coincidentes para cada elemento.	`join … in … on … equals … into …`	Enumerable.GroupJoin Queryable.GroupJoin

Ejemplos de sintaxis de expresiones de consulta

Join

En el ejemplo siguiente se usa la cláusula `join … in … on … equals …` para combinar dos secuencias en función de un valor específico:

```
class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join products and categories based on CategoryId
    var query = from product in products
                join category in categories on product.CategoryId equals category.Id
                select new { product.Name, category.CategoryName };

    foreach (var item in query)
    {
        Console.WriteLine($"{item.Name} - {item.CategoryName}");
    }

    // This code produces the following output:
    //
    // Cola - Beverage
    // Tea - Beverage
    // Apple - Fruit
    // Kiwi - Fruit
    // Carrot - Vegetable
}
```

GroupJoin

En el ejemplo siguiente se usa la cláusula `join … in … on … equals … into …` para combinar dos secuencias en función de un valor específico y se agrupan las coincidencias resultantes de cada elemento:

```
class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //     Cola
    //      Tea
    // Group
    //    Apple
    //     Kiwi
    // Group
    //   Carrot
}
```

Vea también

- System.Linq
- Información general sobre operadores de consulta estándar (C#)
- Tipos anónimos
- Formulación de combinaciones y consultas entre productos
- join (cláusula)

- Join usando claves compuestas
- Procedimiento para combinar contenido de archivos no similares (LINQ) (C#)
- Ordenar los resultados de una cláusula join
- Realizar operaciones de combinación personalizadas
- Realizar combinaciones agrupadas
- Realizar combinaciones internas
- Realizar operaciones de combinación externa izquierda
- Procedimiento para rellenar colecciones de objetos de varios orígenes (LINQ) (C#)

Agrupar datos (C#)

18/03/2020 • 2 minutes to read • Edit Online

El agrupamiento hace referencia a la operación de colocar los datos en grupos de manera que los elementos de cada grupo compartan un atributo común.

La ilustración siguiente muestra los resultados de agrupar una secuencia de caracteres. La clave de cada grupo es el carácter.

Los métodos de operador de consulta estándar que agrupan elementos de datos se enumeran en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN

GroupBy	Agrupa los elementos que comparten un atributo común. Cada grupo se representa mediante un objeto IGrouping<TKey,TElement>.	`group … by`
		o bien
		`group … by … into …`
			Enumerable.GroupBy
			Queryable.GroupBy

ToLookup	Inserta elementos a una Lookup<TKey,TElement> (un diccionario uno a varios) basándose en una función de selector de claves.	No disponible.	Enumerable.ToLookup

Ejemplo de sintaxis de expresiones de consulta

El ejemplo de código siguiente usa la cláusula `group by` para agrupar los enteros de una lista según sean pares o impares.

```
List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199, 329, 446, 208 };

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");
    foreach (int i in group)
        Console.WriteLine(i);
}

/* This code produces the following output:

    Odd numbers:
    35
    3987
    199
    329

    Even numbers:
    44
    200
    84
    4
    446
    208
*/
```

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- group (cláusula)
- Crear grupos anidados
- Procedimiento para agrupar archivos por extensión (LINQ) (C#)
- Agrupar los resultados de consultas
- Realizar una subconsulta en una operación de agrupación
- Procedimiento para dividir un archivo en varios mediante el uso de grupos (LINQ) (C#)

Operaciones de generación (C#)

18/03/2020 • 2 minutes to read • Edit Online

La generación hace referencia a la creación de una nueva secuencia de valores.

Los métodos del operador de consulta estándar que realizan generaciones se indican en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
DefaultIfEmpty	Reemplaza una colección vacía con una colección de singleton con valores predeterminados.	No disponible.	Enumerable.DefaultIfEmpty Queryable.DefaultIfEmpty
Empty	Devuelve una colección vacía.	No disponible.	Enumerable.Empty
Intervalo	Genera una colección que contiene una secuencia de números.	No disponible.	Enumerable.Range
Repeat	Genera una colección que contiene un valor repetido.	No disponible.	Enumerable.Repeat

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))

Operaciones de igualdad (C#)

18/03/2020 • 2 minutes to read • Edit Online

Dos secuencias cuyos respectivos elementos sean iguales y que tengan el mismo número de elementos se consideran iguales.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
SequenceEqual	Determina si dos secuencias son iguales al comparar elementos par a par.	No disponible.	Enumerable.SequenceEqual Queryable.SequenceEqual

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Procedimiento para comparar el contenido de dos carpetas (LINQ) (C#)

Operaciones de elementos (C#)

18/03/2020 • 2 minutes to read • Edit Online

Las operaciones de elementos devuelven un único elemento específico de una secuencia.

Los métodos del operador de consulta estándar que realizan operaciones de elementos se indican en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
ElementAt	Devuelve el elemento situado en un índice especificado de la colección.	No disponible.	Enumerable.ElementAt Queryable.ElementAt
ElementAtOrDefault	Devuelve el elemento situado en un índice especificado de una colección o un valor predeterminado si el índice está fuera del intervalo.	No disponible.	Enumerable.ElementAtOrDefault Queryable.ElementAtOrDefault
First	Devuelve el primer elemento de una colección, o el primer elemento que cumple una condición.	No disponible.	Enumerable.First Queryable.First
FirstOrDefault	Devuelve el primer elemento de una colección, o el primer elemento que cumple una condición. Devuelve un valor predeterminado si dicho elemento no existe.	No disponible.	Enumerable.FirstOrDefault Queryable.FirstOrDefault Queryable.FirstOrDefault<TSource> (IQueryable<TSource>)
Último	Devuelve el último elemento de una colección, o el último elemento que cumple una condición.	No disponible.	Enumerable.Last Queryable.Last
LastOrDefault	Devuelve el último elemento de una colección, o el último elemento que cumple una condición. Devuelve un valor predeterminado si dicho elemento no existe.	No disponible.	Enumerable.LastOrDefault Queryable.LastOrDefault

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Single	Devuelve el único elemento de una colección, o el único elemento que cumple una condición. Se produce un InvalidOperationException si no hay ningún elemento o hay más de un elemento para devolver.	No disponible.	Enumerable.Single Queryable.Single
SingleOrDefault	Devuelve el único elemento de una colección, o el único elemento que cumple una condición. Devuelve un valor predeterminado si no hay ningún elemento para devolver. Se produce un InvalidOperationException si hay más de un elemento para devolver.	No disponible.	Enumerable.SingleOrDefault Queryable.SingleOrDefault

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Procedimiento para buscar el archivo o archivos de mayor tamaño en un árbol de directorios (LINQ) (C#)

Convertir tipos de datos (C#)

18/03/2020 • 4 minutes to read • Edit Online

Los métodos de conversión cambian el tipo de los objetos de entrada.

Las operaciones de conversión en las consultas LINQ son útiles en una serie de aplicaciones. A continuación se muestran algunos ejemplos:

- El método Enumerable.AsEnumerable puede usarse para ocultar una implementación personalizada de tipo de un operador de consulta estándar.
- El método Enumerable.OfType puede usarse para permitir colecciones no parametrizadas para las consultas LINQ.
- Los métodos Enumerable.ToArray, Enumerable.ToDictionary, Enumerable.ToList y Enumerable.ToLookup pueden usarse para aplicar la ejecución de consultas inmediata en lugar de aplazarla hasta que se enumere la consulta.

Métodos

En la siguiente tabla se muestran los métodos de operadores de consulta estándar que efectúan conversiones de tipo de datos.

Los métodos de conversión de esta tabla cuyos nombres comienzan por "As" cambian el tipo estático de la colección de origen, pero no lo enumeran. Los métodos cuyos nombres empiezan por "To" enumeran la colección de origen y colocan los elementos en el tipo de colección correspondiente.

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
AsEnumerable	Devuelve la entrada con tipo como IEnumerable<T>.	No disponible.	Enumerable.AsEnumerable
AsQueryable	Convierte un IEnumerable (genérico) en un IQueryable (genérico).	No disponible.	Queryable.AsQueryable
Conversión de tipos explícita	Convierte los elementos de una colección en un tipo especificado.	Use una variable de rango con tipo explícito. Por ejemplo: `from string str in words`	Enumerable.Cast Queryable.Cast
OfType	Filtra valores en función de su capacidad para convertirse en un tipo especificado.	No disponible.	Enumerable.OfType Queryable.OfType
ToArray	Convierte una colección en una matriz. Este método fuerza la ejecución de la consulta.	No disponible.	Enumerable.ToArray

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
ToDictionary	Coloca elementos en Dictionary<TKey,TValue> basándose en una función de selector de claves. Este método fuerza la ejecución de la consulta.	No disponible.	Enumerable.ToDictionary
ToList	Convierte una colección en List<T>. Este método fuerza la ejecución de la consulta.	No disponible.	Enumerable.ToList
ToLookup	Coloca elementos en una Lookup<TKey,TElement> (un diccionario uno a varios) basándose en una función de selector de claves. Este método fuerza la ejecución de la consulta.	No disponible.	Enumerable.ToLookup

Ejemplo de sintaxis de expresiones de consulta

En el ejemplo de código siguiente se usa una variable de rango con tipo explícito para convertir un tipo en un subtipo antes de obtener acceso a un miembro que solo está disponible en el subtipo.

```
class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
                where cPlant.TrapType == "Snap Trap"
                select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:

        Venus Fly Trap
        Waterwheel Plant
    */
}
```

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- from (cláusula)
- Expresiones de consulta LINQ
- Procedimiento para consultar un objeto ArrayList con LINQ (C#)

Operaciones de concatenación (C#)

18/03/2020 • 2 minutes to read • Edit Online

La concatenación hace referencia a la operación de anexar una secuencia a otra.

En la siguiente ilustración se muestra una operación de concatenación en dos secuencias de caracteres.

Los métodos del operador de consulta estándar que efectúan una concatenación se indican en la siguiente sección.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Concat	Concatena dos secuencias para formar una secuencia.	No disponible.	Enumerable.Concat Queryable.Concat

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Procedimiento para combinar y comparar colecciones de cadenas (LINQ) (C#)

Operaciones de agregación (C#)

18/03/2020 • 2 minutes to read • Edit Online

Una operación de agregación calcula un valor único a partir de una colección de valores. Un ejemplo de operación de agregación es calcular el promedio de temperatura diaria a partir de los valores de temperatura diaria durante un mes.

En la siguiente ilustración se muestran los resultados de dos operaciones de agregación diferentes en una secuencia de números. La primera operación suma los números. La segunda operación devuelve el valor máximo de la secuencia.

Los métodos del operador de consulta estándar que realizan operaciones de agregación se indican en la sección siguiente.

Métodos

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Agregar	Realiza una operación de agregación personalizada en los valores de una colección.	No disponible.	Enumerable.Aggregate Queryable.Aggregate
Average	Calcula el valor medio de una colección de valores.	No disponible.	Enumerable.Average Queryable.Average
Recuento	Cuenta los elementos de una colección; opcionalmente, solo aquellos que satisfacen una función de predicado.	No disponible.	Enumerable.Count Queryable.Count
LongCount	Cuenta los elementos de una gran colección; opcionalmente, solo aquellos que satisfacen una función de predicado.	No disponible.	Enumerable.LongCount Queryable.LongCount
Máx.	Determina el valor máximo de una colección.	No disponible.	Enumerable.Max Queryable.Max

NOMBRE DEL MÉTODO	DESCRIPTION	SINTAXIS DE LA EXPRESIÓN DE CONSULTA DE C#	MÁS INFORMACIÓN
Mín.	Determina el valor mínimo de una colección.	No disponible.	Enumerable.Min Queryable.Min
Sum	Calcula la suma de los valores de una colección.	No disponible.	Enumerable.Sum Queryable.Sum

Vea también

- System.Linq
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Procedimiento para calcular valores de columna en un archivo de texto CSV (LINQ) (C#)
- Procedimiento para buscar el archivo o archivos de mayor tamaño en un árbol de directorios (LINQ) (C#)
- Procedimiento para buscar el número total de bytes de un conjunto de carpetas (LINQ) (C#)

LINQ to Objects (C#)

29/06/2020 • 3 minutes to read • Edit Online

El término "LINQ to Objects" se refiere al uso de consultas LINQ con cualquier colección IEnumerable o IEnumerable<T> directamente, sin usar un proveedor o una API de LINQ intermedios como LINQ to SQL o LINQ to XML. Puede usar LINQ para consultar cualquier colección enumerable, como List<T>, Array o Dictionary<TKey,TValue>. La colección puede haberla definido el usuario, o bien puede que la haya devuelto una API de .NET.

Básicamente, LINQ to Objects representa un nuevo enfoque para las colecciones. En el sistema antiguo, tenía que escribir complejos bucles `foreach` que especificaban cómo recuperar los datos de una colección. En el enfoque de LINQ, se escribe código declarativo que describe qué se quiere recuperar.

Además, las consultas LINQ ofrecen tres ventajas principales respecto a los bucles `foreach` tradicionales:

- Son más concisas y legibles, especialmente cuando se filtran varias condiciones.
- Proporcionan funcionalidades eficaces para filtrar, ordenar y agrupar con un código de aplicación mínimo.
- Se pueden migrar a otros orígenes de datos con muy poca o ninguna modificación.

Por lo general, cuanto más compleja es la operación que se quiere realizar en los datos, más ventajas se obtienen al usar LINQ en lugar de las técnicas de iteración tradicionales.

El propósito de esta sección es mostrar el enfoque de LINQ con algunos ejemplos seleccionados. No pretende ser exhaustiva.

En esta sección

LINQ y cadenas (C#)
Explica cómo se puede usar LINQ para consultar y transformar cadenas y colecciones de cadenas. También incluye vínculos a artículos que muestran estos principios.

LINQ y reflexión (C#)
Vincula a un ejemplo que muestra cómo usa LINQ la reflexión.

LINQ y directorios de archivos (C#)
Explica cómo se puede usar LINQ para interactuar con sistemas de archivos. También incluye vínculos a artículos que muestran estos conceptos.

Procedimiento para consultar un objeto ArrayList con LINQ (C#)
Muestra cómo consultar un objeto ArrayList en C#.

Procedimiento para agregar métodos personalizados para las consultas LINQ (C#)
Explica cómo extender el conjunto de métodos que puede usar para consultas LINQ agregando métodos de extensión a la interfaz IEnumerable<T>.

Language Integrated Query (LINQ) (C#)
Proporciona vínculos a artículos que explican LINQ e incluye ejemplos de código que realizan consultas.

LINQ y cadenas (C#)

18/03/2020 • 5 minutes to read • Edit Online

Se puede usar LINQ para consultar y transformar cadenas y colecciones de cadenas. Puede resultar especialmente útil con datos semiestructurados de archivos de texto. Las consultas LINQ se pueden combinar con funciones de cadena tradicionales y expresiones regulares. Por ejemplo, puede usar el método String.Split o Regex.Split para crear una matriz de cadenas que después puede consultar o modificar mediante LINQ. Puede usar el método Regex.IsMatch en la cláusula `where` de una consulta LINQ. Y puede usar LINQ para consultar o modificar los resultados MatchCollection que se han devuelto mediante una expresión regular.

También puede usar las técnicas descritas en esta sección para transformar datos de texto semiestructurados en XML. Para más información, consulte Procedimiento para generar XML a partir de archivos CSV (C#).

Los ejemplos de esta sección se dividen en dos categorías:

Consulta de un bloque de texto

Puede consultar, analizar y modificar bloques de texto dividiéndolos en una matriz consultable de cadenas más pequeñas mediante el método String.Split o el método Regex.Split. Puede dividir el texto de origen en palabras, frases, párrafos, páginas o cualquier otro criterio y luego realizar divisiones adicionales si son necesarias en la consulta.

- Procedimiento para realizar un recuento de las repeticiones de una palabra en una cadena (LINQ) (C#)
  Muestra cómo usar LINQ para consultas simples en texto.
- Procedimiento para buscar frases que contengan un conjunto especificado de palabras (LINQ) (C#)

  Muestra cómo dividir archivos de texto en límites arbitrarios y cómo realizar consultas en cada parte.
- Procedimiento para buscar caracteres en una cadena (LINQ) (C#)

  Muestra que una cadena es un tipo que se puede consultar.
- Procedimiento para combinar consultas LINQ con expresiones regulares (C#)

  Muestra cómo usar expresiones regulares en consultas LINQ para la coincidencia de patrones compleja en resultados de consulta filtrados.

Consulta de datos semiestructurados en formato de texto

Muchos tipos diferentes de archivos de texto se componen de una serie de líneas, a menudo con un formato similar, como archivos delimitados por comas o líneas de longitud fija. Después de leer uno de estos archivos de texto en la memoria, puede usar LINQ para consultar o modificar las líneas. Las consultas LINQ también simplifican la tarea de combinar datos de varios orígenes.

- Procedimiento para buscar la diferencia de conjuntos entre dos listas (LINQ) (C#)

  Muestra cómo encontrar todas las cadenas que se encuentran en una lista pero no en la otra.
- Procedimiento para ordenar o filtrar datos de texto por palabra o campo (LINQ) (C#)

  Muestra cómo ordenar líneas de texto según cualquier palabra o campo.
- Procedimiento para reordenar los campos de un archivo delimitado (LINQ) (C#)

Muestra cómo cambiar el orden de los campos en una línea en un archivo .csv.

- Procedimiento para combinar y comparar colecciones de cadenas (LINQ) (C#)

  Muestra cómo combinar listas de cadenas de varias maneras.

- Procedimiento para rellenar colecciones de objetos de varios orígenes (LINQ) (C#)

  Muestra cómo crear colecciones de objetos mediante varios archivos de texto como orígenes de datos.

- Procedimiento para combinar contenido de archivos no similares (LINQ) (C#)

  Muestra cómo combinar cadenas de dos listas en una sola mediante una clave coincidente.

- Procedimiento para dividir un archivo en varios mediante el uso de grupos (LINQ) (C#)

  Muestra cómo crear nuevos archivos mediante un solo archivo como origen de datos.

- Procedimiento para calcular valores de columna en un archivo de texto CSV (LINQ) (C#)

  Muestra cómo realizar cálculos matemáticos en datos de texto en archivos .csv.

Vea también

- Language Integrated Query (LINQ) (C#)
- Procedimiento para generar XML a partir de archivos CSV

Procedimiento para realizar un recuento de las repeticiones de una palabra en una cadena (LINQ) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo usar una consulta LINQ para contar las apariciones de una palabra determinada en una cadena. Observe que para realizar el recuento, primero se llama al método Split para crear una matriz de palabras. Existe un costo de rendimiento en el método Split. Si la única operación de la cadena es para contar las palabras, debe considerar la posibilidad de usar en su lugar los métodos Matches o IndexOf. Pero si el rendimiento no es un problema crítico, o si ya ha dividido la frase para realizar otros tipos de consultas, tiene sentido usar LINQ para además contar las palabras o frases.

Ejemplo

```
class CountWords
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects" +
          @" have not been well integrated. Programmers work in C# or Visual Basic" +
          @" and also in SQL or XQuery. On the one side are concepts such as classes," +
          @" objects, fields, inheritance, and .NET Framework APIs. On the other side" +
          @" are tables, columns, rows, nodes, and separate languages for dealing with" +
          @" them. Data types often require translation between the two worlds; there are" +
          @" different standard functions. Because the object world has no notion of query, a" +
          @" query can only be represented as a string without compile-time type checking or" +
          @" IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to" +
          @" objects in memory is often tedious and error-prone.";

        string searchTerm = "data";

        //Convert the string into an array of words
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
StringSplitOptions.RemoveEmptyEntries);

        // Create the query.  Use ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
                         where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
                         select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrences(s) of the search term \"{1}\" were found.", wordCount, searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)

Procedimiento para buscar frases que contengan un conjunto especificado de palabras (LINQ) (C#)

18/03/2020 • 3 minutes to read • Edit Online

En este ejemplo se muestra cómo buscar frases en un archivo de texto que contengan coincidencias con cada uno de los conjuntos de palabras especificados. Aunque la matriz de términos de búsqueda está codificada de forma rígida en este ejemplo, también se podría rellenar dinámicamente en tiempo de ejecución. En este ejemplo, la consulta devuelve las frases que contienen las palabras "Historically", "data" e "integrated".

Ejemplo

```
class FindSentences
{
    static void Main()
    {
        string text = @"Historically, the world of data and the world of objects " +
        @"have not been well integrated. Programmers work in C# or Visual Basic " +
        @"and also in SQL or XQuery. On the one side are concepts such as classes, " +
        @"objects, fields, inheritance, and .NET Framework APIs. On the other side " +
        @"are tables, columns, rows, nodes, and separate languages for dealing with " +
        @"them. Data types often require translation between the two worlds; there are " +
        @"different standard functions. Because the object world has no notion of query, a " +
        @"query can only be represented as a string without compile-time type checking or " +
        @"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to " +
        @"objects in memory is often tedious and error-prone.";

        // Split the text block into an array of sentences.
        string[] sentences = text.Split(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically populated at runtime.
        string[] wordsToMatch = { "Historically", "data", "integrated" };

        // Find sentences that contain all the terms in the wordsToMatch array.
        // Note that the number of terms to match is not specified at compile time.
        var sentenceQuery = from sentence in sentences
                            let w = sentence.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                                                   StringSplitOptions.RemoveEmptyEntries)
                            where w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
                            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
Historically, the world of data and the world of objects have not been well integrated
*/
```

La consulta primero divide el texto en frases y, luego, divide las frases en una matriz de cadenas que contienen

cada palabra. Para cada una de estas matrices, el método Distinct quita todas las palabras duplicadas y, después, la consulta realiza una operación Intersect en la matriz de palabras y en la matriz `wordsToMatch`. Si el recuento de la intersección es igual que el recuento de la matriz `wordsToMatch`, se han encontrado todas las palabras y se devuelve la frase original.

En la llamada a Split, los signos de puntuación se usan como separadores para quitar las frases de la cadena. Si no lo hiciera podría tener, por ejemplo, una cadena "Historically", lo que no coincidiría con el "Historically" de la matriz `wordsToMatch`. Podría tener que usar separadores adicionales, en función de los tipos de puntuación del texto de origen.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)

Procedimiento para buscar caracteres en una cadena (LINQ) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Como la clase String implementa la interfaz IEnumerable<T> genérica, cualquier cadena puede consultarse como una secuencia de caracteres. Pero esto no es un uso habitual de LINQ. Para operaciones de coincidencia de patrones complejas, use la clase Regex.

Ejemplo

En el ejemplo siguiente se consulta una cadena para determinar el número de dígitos numéricos que contiene. Tenga en cuenta que la consulta se "reutiliza" después de que se ejecute la primera vez. Esto es posible porque la propia consulta no almacena ningún resultado real.

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> stringQuery =
          from ch in aString
          where Char.IsDigit(ch)
          select ch;

        // Execute the query
        foreach (char c in stringQuery)
            Console.Write(c + " ");

        // Call the Count method on the existing query.
        int count = stringQuery.Count();
        Console.WriteLine("Count = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
  Output: 9 9 7 4 1 2 8 9
  Count = 8
  ABCDE99F
*/
```

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)
- Procedimiento para combinar consultas LINQ con expresiones regulares (C#)

Procedimiento para combinar consultas LINQ con expresiones regulares (C#)

18/03/2020 • 3 minutes to read • Edit Online

En este ejemplo se muestra cómo usar la clase Regex para crear una expresión regular para coincidencias más complejas en cadenas de texto. Con la consulta LINQ, resulta fácil filtrar por los archivos exactos que se quieren buscar con la expresión regular y dar forma a los resultados.

Ejemplo

```
class QueryWithRegEx
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio 14.0\";
        // One of the following paths may be more appropriate on your computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual (Basic|C#|C\+\+|Studio)");

        // Search the contents of each .htm file.
        // Remove the where clause to find even more matchedValues!
        // This query produces a list of files where a match
        // was found, and a list of the matchedValues in that file.
        // Note: Explicit typing of "Match" in select clause.
        // This is required because MatchCollection is not a
        // generic IEnumerable collection.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = System.IO.File.ReadAllText(file.FullName)
            let matches = searchTerm.Matches(fileText)
            where matches.Count > 0
            select new
            {
                name = file.FullName,
                matchedValues = from System.Text.RegularExpressions.Match match in matches
                                select match.Value
            };

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm.ToString());

        foreach (var v in queryMatchingFiles)
        {
            // Trim the path a bit, then write
            // the file name in which a match was found.
            string s = v.name.Substring(startFolder.Length - 1);
            Console.WriteLine(s);

            // For this file, write out all the matching strings
            foreach (var v2 in v.matchedValues)
            {
```

```
                Console.WriteLine("  " + v2);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // This method assumes that the application has discovery
    // permissions for all folders under the specified path.
    static IEnumerable<System.IO.FileInfo> GetFiles(string path)
    {
        if (!System.IO.Directory.Exists(path))
            throw new System.IO.DirectoryNotFoundException();

        string[] fileNames = null;
        List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

        fileNames = System.IO.Directory.GetFiles(path, "*.*", System.IO.SearchOption.AllDirectories);
        foreach (string name in fileNames)
        {
            files.Add(new System.IO.FileInfo(name));
        }
        return files;
    }
}
```

Tenga en cuenta que también puede consultar el objeto MatchCollection devuelto por una búsqueda `RegEx`. En este ejemplo se genera solo el valor de cada coincidencia en los resultados. Pero también es posible usar LINQ para realizar todo tipo de filtrado, ordenación y agrupación en esa colección. Dado que MatchCollection no es una colección genérica IEnumerable, tendrá que indicar explícitamente el tipo de la variable de rango en la consulta.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para buscar la diferencia de conjuntos entre dos listas (LINQ) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo usar LINQ para comparar dos listas de cadenas y generar estas líneas, que están en names1.txt pero no en names2.txt.

Para crear los archivos de datos

1. Copie names1.txt y names2.txt en la carpeta de la solución, como se muestra en Procedimiento para combinar y comparar colecciones de cadenas (LINQ) (C#).

Ejemplo

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> differenceQuery =
          names1.Except(names2);

        // Execute the query.
        Console.WriteLine("The following lines are in names1.txt but not names2.txt");
        foreach (string s in differenceQuery)
            Console.WriteLine(s);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
     The following lines are in names1.txt but not names2.txt
    Potra, Cristina
    Noriega, Fabricio
    Aw, Kam Foo
    Toyoshima, Tim
    Guy, Wey Yuan
    Garcia, Debra
     */
```

Algunos tipos de operaciones de consulta en C#, como Except, Distinct, Union y Concat, solo pueden expresarse en una sintaxis basada en método.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)

Procedimiento para ordenar o filtrar datos de texto por palabra o campo (LINQ) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En el ejemplo siguiente se muestra cómo ordenar líneas de texto estructurado, como valores separados por comas, por cualquier campo de la línea. El campo se puede especificar dinámicamente en tiempo de ejecución. Supongamos que los campos de scores.csv representan el número de identificación de un alumno, seguido de una serie de cuatro calificaciones.

Para crear un archivo que contenga datos

1. Copie los datos de scores.csv del tema Procedimiento para combinar contenido de archivos no similares (LINQ) (C#).

Ejemplo

```
public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:", sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                         let fields = line.Split(',')
                         orderby fields[num] descending
                         select line;

        return scoreQuery;
    }
}
/* Output (if sortField == 1):
   Sorted highest to lowest by field [1]:
    116, 99, 86, 90, 94
    120, 99, 82, 81, 79
    111, 97, 92, 81, 60
    114, 97, 89, 85, 82
    121, 96, 85, 91, 60
    122, 94, 92, 91, 91
    117, 93, 92, 80, 87
    118, 92, 90, 83, 78
    113, 88, 94, 65, 91
    112, 75, 84, 91, 39
    119, 68, 79, 88, 92
    115, 35, 72, 91, 70
 */
```

En este ejemplo también se muestra cómo devolver una variable de consulta desde un método.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)

Procedimiento para reordenar los campos de un archivo delimitado (LINQ) (C#)

18/03/2020 • 3 minutes to read • Edit Online

Un archivo de valores separados por comas (CSV) es un archivo de texto que se usa a menudo para almacenar datos de hoja de cálculo u otros datos tabulares que se representan mediante filas y columnas. Mediante el uso del método Split para separar los campos, es muy fácil consultar y manipular los archivos CSV con LINQ. De hecho, la misma técnica puede usarse para reordenar los elementos de cualquier línea estructurada de texto, no se limita a un archivo CSV.

En el ejemplo siguiente, suponga que las tres columnas representan el "apellido", el "nombre" y el "ID" de los alumnos. Los campos están en orden alfabético según el apellido de los alumnos. La consulta genera una nueva secuencia en la que la columna de identificador aparece en primer lugar, seguida por una segunda columna que combina el nombre y el apellido del alumno. Las líneas se reordenan según el campo ID. Los resultados se guardan en un archivo nuevo y no se modifican los datos originales.

Para crear el archivo de datos

1. Copie las líneas siguientes en un archivo de texto sin formato que se denomine spreadsheet1.csv. Guarde el archivo en la carpeta del proyecto.

```
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
Tucker,Michael,122
Zabokritski,Eugene,121
```

Ejemplo

```
class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines = System.IO.File.ReadAllLines(@"../../../spreadsheet1.csv");

        // Create the query. Put field 2 first, then
        // reverse and combine fields 0 and 1 from the old field
        IEnumerable<string> query =
            from line in lines
            let x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../spreadsheet2.csv", query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key to exit");
        Console.ReadKey();
    }
}
/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
 */
```

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])
- Procedimiento para generar XML a partir de archivos CSV (C#)

Procedimiento para combinar y comparar colecciones de cadenas (LINQ) (C#)

18/03/2020 • 3 minutes to read • Edit Online

En este ejemplo se muestra cómo combinar archivos que contienen líneas de texto y después ordenar los resultados. En concreto, se muestra cómo realizar una concatenación simple, una unión y una intersección en los dos conjuntos de líneas de texto.

Para configurar el proyecto y los archivos de texto

1. Copie estos nombres en un archivo de texto denominado names1.txt y guárdelo en la carpeta del proyecto:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copie estos nombres en un archivo de texto denominado names2.txt y guárdelo en la carpeta del proyecto. Tenga en cuenta que los dos archivos tienen algunos nombres en común.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Ejemplo

```
class MergeStrings
    {
        static void Main(string[] args)
        {
            //Put text files in your solution folder
            string[] fileA = System.IO.File.ReadAllLines(@"../../../names1.txt");
            string[] fileB = System.IO.File.ReadAllLines(@"../../../names2.txt");

            //Simple concatenation and sort. Duplicates are preserved.
            IEnumerable<string> concatQuery =
                fileA.Concat(fileB).OrderBy(s => s);

            // Pass the query variable to another function for execution.
            OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are preserved:");

            // Concatenate and remove duplicate names based on
            // default string comparer.
```

```
        // default string comparer.
        IEnumerable<string> uniqueNamesQuery =
            fileA.Union(fileB).OrderBy(s => s);
        OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

        // Find the names that occur in both files (based on
        // default string comparer).
        IEnumerable<string> commonNamesQuery =
            fileA.Intersect(fileB);
        OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

        // Find the matching fields in each list. Merge the two
        // results by using Concat, and then
        // sort using the default string comparer.
        string nameMatch = "Garcia";

        IEnumerable<String> tempQuery1 =
            from name in fileA
            let n = name.Split(',')
            where n[0] == nameMatch
            select name;

        IEnumerable<string> tempQuery2 =
            from name2 in fileB
            let n2 = name2.Split(',')
            where n2[0] == nameMatch
            select name2;

        IEnumerable<string> nameMatchQuery =
            tempQuery1.Concat(tempQuery2).OrderBy(s => s);
        OutputQueryResults(nameMatchQuery, $"Concat based on partial name match \"{nameMatch}\":");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void OutputQueryResults(IEnumerable<string> query, string message)
    {
        Console.WriteLine(System.Environment.NewLine + message);
        foreach (string item in query)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("{0} total names in list", query.Count());
    }
}
/* Output:
    Simple concatenate and sort. Duplicates are preserved:
    Aw, Kam Foo
    Bankov, Peter
    Bankov, Peter
    Beebe, Ann
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
    Guy, Wey Yuan
    Holm, Michael
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    20 total names in list
```

```
        26 total names in list

        Union removes duplicate names:
        Aw, Kam Foo
        Bankov, Peter
        Beebe, Ann
        El Yassir, Mehdi
        Garcia, Debra
        Garcia, Hugo
        Giakoumakis, Leo
        Gilchrist, Beth
        Guy, Wey Yuan
        Holm, Michael
        Liu, Jinghao
        McLin, Nkenge
        Myrcha, Jacek
        Noriega, Fabricio
        Potra, Cristina
        Toyoshima, Tim
        16 total names in list

        Merge based on intersect:
        Bankov, Peter
        Holm, Michael
        Garcia, Hugo
        Beebe, Ann
        4 total names in list

        Concat based on partial name match "Garcia":
        Garcia, Debra
        Garcia, Hugo
        Garcia, Hugo
        3 total names in list
*/
```

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para rellenar colecciones de objetos de varios orígenes (LINQ) (C#)

18/03/2020 • 5 minutes to read • Edit Online

En este ejemplo se muestra cómo combinar datos de orígenes diferentes en una secuencia de tipos nuevos.

> **NOTE**
>
> No intente unir datos en memoria o datos del sistema de archivos con datos que todavía están en una base de datos. Dichas combinaciones entre dominios pueden producir resultados indefinidos porque hay diferentes maneras de definir las operaciones de combinación para las consultas de base de datos y otros tipos de orígenes. Además, existe el riesgo de que esta operación produzca una excepción de memoria insuficiente si la cantidad de datos existente en la base de datos es considerable. Para combinar datos de una base de datos con datos en memoria, primero debe llamar a `ToList` o a `ToArray` en la base de datos de consulta y, luego, debe efectuar la combinación en la colección devuelta.

Para crear el archivo de datos

Copie los archivos names.csv y scores.csv en la carpeta del proyecto, como se describe en Procedimiento para combinar contenido de archivos no similares (LINQ) (C#).

Ejemplo

En el ejemplo siguiente se muestra cómo usar un tipo `Student` con nombre para almacenar los datos combinados de dos colecciones de cadenas en memoria que simulan datos de hoja de cálculo en formato .csv. La primera colección de cadenas representa los nombres y los identificadores de los estudiantes, mientras que la segunda colección representa el identificador de los estudiantes (en la primera columna) y cuatro notas de exámenes. El identificador se usa como clave externa.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to join content from
        // dissimilar files (LINQ).

        // Each line of names.csv consists of a last name, a first name, and an
        // ID number, separated by commas. For example, Omelchenko,Svetlana,111
        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");

        // Each line of scores.csv consists of an ID number and four test
        // scores, separated by commas. For example, 111, 97, 92, 81, 60
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");
```

```
        // Merge the data sources using a named type.
        // var could be used instead of an explicit type. Note the dynamic
        // creation of a list of ints for the ExamScores member. The first item
        // is skipped in the split string because it is the student ID,
        // not an exam score.
        IEnumerable<Student> queryNamesScores =
            from nameLine in names
            let splitName = nameLine.Split(',')
            from scoreLine in scores
            let splitScoreLine = scoreLine.Split(',')
            where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
            select new Student()
            {
                FirstName = splitName[0],
                LastName = splitName[1],
                ID = Convert.ToInt32(splitName[2]),
                ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                              select Convert.ToInt32(scoreAsText)).
                              ToList()
            };

        // Optional. Store the newly created student objects in memory
        // for faster access in future queries. This could be useful with
        // very large data files.
        List<Student> students = queryNamesScores.ToList();

        // Display each student's name and exam score average.
        foreach (var student in students)
        {
            Console.WriteLine("The average score of {0} {1} is {2}.",
                student.FirstName, student.LastName,
                student.ExamScores.Average());
        }

        //Keep console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    The average score of Omelchenko Svetlana is 82.5.
    The average score of O'Donnell Claire is 72.25.
    The average score of Mortensen Sven is 84.5.
    The average score of Garcia Cesar is 88.25.
    The average score of Garcia Debra is 67.
    The average score of Fakhouri Fadi is 92.25.
    The average score of Feng Hanying is 88.
    The average score of Garcia Hugo is 85.75.
    The average score of Tucker Lance is 81.75.
    The average score of Adams Terry is 85.25.
    The average score of Zabokritski Eugene is 83.
    The average score of Tucker Michael is 92.
 */
```

En la cláusula select se usa un inicializador de objeto para crear una instancia de cada objeto `Student` nuevo usando los datos de los dos orígenes.

Si no tiene que almacenar los resultados de una consulta, los tipos anónimos pueden ser más convenientes que los tipos con nombre. Los tipos con nombre son necesarios si pasa los resultados de la consulta fuera del método en el que se ejecuta la consulta. En el ejemplo siguiente se ejecuta la misma tarea que en el ejemplo anterior, con la diferencia de que se usan tipos anónimos en lugar de tipos con nombre:

```
// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 =
    from nameLine in names
    let splitName = nameLine.Split(',')
    from scoreLine in scores
    let splitScoreLine = scoreLine.Split(',')
    where Convert.ToInt32(splitName[2]) == Convert.ToInt32(splitScoreLine[0])
    select new
    {
        First = splitName[0],
        Last = splitName[1],
        ExamScores = (from scoreAsText in splitScoreLine.Skip(1)
                      select Convert.ToInt32(scoreAsText))
                      .ToList()
    };

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine("The average score of {0} {1} is {2}.",
        student.First, student.Last, student.ExamScores.Average());
}
```

Vea también

- LINQ y cadenas (C#)
- Inicializadores de objeto y colección
- Tipos anónimos

Procedimiento para dividir un archivo en muchos mediante grupos (LINQ) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra una manera de combinar el contenido de dos archivos y luego crear un conjunto de archivos nuevos que organicen los datos de una forma nueva.

Para crear los archivos de datos

1. Copie estos nombres en un archivo de texto denominado names1.txt y guárdelo en la carpeta del proyecto:

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. Copie estos nombres en un archivo de texto denominado names2.txt y guárdelo en la carpeta del proyecto: tenga en cuenta que los dos archivos tienen algunos nombres en común.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

Ejemplo

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA = System.IO.File.ReadAllLines(@"../../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        var mergeQuery = fileA.Union(fileB);

        // Group the names by the first letter in the last name.
        var groupQuery = from name in mergeQuery
                         let n = name.Split(',')
                         group name by n[0][0] into g
                         orderby g.Key
                         select g;
```

```
        // Create a new file for each group that was created
        // Note that nested foreach loops are required to access
        // individual items with each group.
        foreach (var g in groupQuery)
        {
            // Create the new file name.
            string fileName = @"../../../testFile_" + g.Key + ".txt";

            // Output to display.
            Console.WriteLine(g.Key);

            // Write file.
            using (System.IO.StreamWriter sw = new System.IO.StreamWriter(fileName))
            {
                foreach (var item in g)
                {
                    sw.WriteLine(item);
                    // Output to console for example purposes.
                    Console.WriteLine("   {0}", item);
                }
            }
        }
        // Keep console window open in debug mode.
        Console.WriteLine("Files have been written. Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
    A
       Aw, Kam Foo
    B
       Bankov, Peter
       Beebe, Ann
    E
       El Yassir, Mehdi
    G
       Garcia, Hugo
       Guy, Wey Yuan
       Garcia, Debra
       Gilchrist, Beth
       Giakoumakis, Leo
    H
       Holm, Michael
    L
       Liu, Jinghao
    M
       Myrcha, Jacek
       McLin, Nkenge
    N
       Noriega, Fabricio
    P
       Potra, Cristina
    T
       Toyoshima, Tim
 */
```

El programa escribe un archivo independiente para cada grupo en la misma carpeta que los archivos de datos.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para combinar contenido de archivos no similares (LINQ) (C#)

18/03/2020 • 3 minutes to read • Edit Online

En este ejemplo se muestra cómo combinar datos de dos archivos delimitados por comas que comparten un valor común que se usa como clave coincidente. Esta técnica puede ser útil si tiene que combinar datos de dos hojas de cálculo o si tiene que combinar en un archivo nuevo datos procedentes de una hoja de cálculo y de un archivo que tiene otro formato. Puede modificar el ejemplo para adaptarlo a cualquier tipo de texto estructurado.

Para crear los archivos de datos

1. Copie las líneas siguientes en un archivo llamado *scores.csv* y guárdelo en la carpeta del proyecto. El archivo representa datos de una hoja de cálculo. La columna 1 es el identificador del estudiante y las columnas comprendidas entre la 2 y la 5 son las notas de las pruebas.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

2. Copie las líneas siguientes en un archivo llamado *names.csv* y guárdelo en la carpeta del proyecto. El archivo representa una hoja de cálculo que contiene el nombre, los apellidos y el identificador de los estudiantes.

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Ejemplo

```
using System;
using System.Collections.Generic;
using System.Linq;

class JoinStrings
{
```

```
    static void Main()
    {
        // Join content from dissimilar files that contain
        // related information. File names.csv contains the student
        // name plus an ID number. File scores.csv contains the ID
        // and a set of four test scores. The following query joins
        // the scores to the student names by using ID as a
        // matching key.

        string[] names = System.IO.File.ReadAllLines(@"../../../names.csv");
        string[] scores = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Name:    Last[0],       First[1],  ID[2]
        //          Omelchenko,    Svetlana,  11
        // Score:   StudentID[0],  Exam1[1]   Exam2[2],  Exam3[3],  Exam4[4]
        //          111,           97,        92,        81,        60

        // This query joins two dissimilar spreadsheets based on common ID value.
        // Multiple from clauses are used instead of a join clause
        // in order to store results of id.Split.
        IEnumerable<string> scoreQuery1 =
            from name in names
            let nameFields = name.Split(',')
            from id in scores
            let scoreFields = id.Split(',')
            where Convert.ToInt32(nameFields[2]) == Convert.ToInt32(scoreFields[0])
            select nameFields[0] + "," + scoreFields[1] + "," + scoreFields[2]
                   + "," + scoreFields[3] + "," + scoreFields[4];

        // Pass a query variable to a method and execute it
        // in the method. The query itself is unchanged.
        OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void OutputQueryResults(IEnumerable<string> query, string message)
    {
        Console.WriteLine(System.Environment.NewLine + message);
        foreach (string item in query)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine("{0} total names in list", query.Count());
    }
}
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
 */
```

Vea también

- LINQ y cadenas (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para calcular valores de columna en un archivo de texto CSV (LINQ) (C#)

18/03/2020 • 4 minutes to read • Edit Online

En este ejemplo se muestra cómo efectuar cálculos agregados (como sumas, promedios, mínimos y máximos) en las columnas de un archivo .csv. Los principios de ejemplo que se muestran aquí se pueden aplicar a otros tipos de textos estructurados.

Para crear el archivo de origen

1. Copie las líneas siguientes en un archivo llamado scores.csv y guárdelo en la carpeta del proyecto. Imagínese que la primera columna representa un identificador de estudiante y que las columnas siguientes representan las notas de cuatro exámenes.

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

Ejemplo

```
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines = System.IO.File.ReadAllLines(@"../../../scores.csv");

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");
```

```
        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
        // passed in dynamically at runtime.

        // Variable columnQuery is an IEnumerable<int>.
        // The following query performs two steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) convert the element at position examNum to an int
        //    and select it.
        var columnQuery =
            from line in strs
            let elements = line.Split(',')
            select Convert.ToInt32(elements[examNum]);

        // Execute the query and cache the results to improve
        // performance. This is helpful only with very large files.
        var results = columnQuery.ToList();

        // Perform aggregate calculations Average, Max, and
        // Min on the column specified by examNum.
        double average = results.Average();
        int max = results.Max();
        int min = results.Min();

        Console.WriteLine("Exam #{0}: Average:{1:##.##} High Score:{2} Low Score:{3}",
                  examNum, average, max, min);
    }

    static void MultiColumns(IEnumerable<string> strs)
    {
        Console.WriteLine("Multi Column Query:");

        // Create a query, multiColQuery. Explicit typing is used
        // to make clear that, when executed, multiColQuery produces
        // nested sequences. However, you get the same results by
        // using 'var'.

        // The multiColQuery query performs the following steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) use Skip to skip the "Student ID" column, and store the
        //    rest of the row in scores.
        // 3) convert each score in the current row from a string to
        //    an int, and select that entire sequence as one row
        //    in the results.
        IEnumerable<IEnumerable<int>> multiColQuery =
            from line in strs
            let elements = line.Split(',')
            let scores = elements.Skip(1)
            select (from str in scores
                    select Convert.ToInt32(str));

        // Execute the query and cache the results to improve
        // performance.
        // ToArray could be used instead of ToList.
        var results = multiColQuery.ToList();

        // Find out how many columns you have in results.
        int columnCount = results[0].Count();

        // Perform aggregate calculations Average, Max, and
        // Min on each column.
        // Perform one iteration of the loop for each column
        // of scores.
        // You can use a for loop instead of a foreach loop
        // because you already executed the multiColQuery
        // query by calling ToList.
        for (int column = 0; column < columnCount; column++)
```

```
        for (int column = 0; column < columnCount; column++)
        {
            var results2 = from row in results
                           select row.ElementAt(column);
            double average = results2.Average();
            int max = results2.Max();
            int min = results2.Min();

            // Add one to column because the first exam is Exam #1,
            // not Exam #0.
            Console.WriteLine("Exam #{0} Average: {1:##.##} High Score: {2} Low Score: {3}",
                          column + 1, average, max, min);
        }
    }
}
/* Output:
    Single Column Query:
    Exam #4: Average:76.92 High Score:94 Low Score:39

    Multi Column Query:
    Exam #1 Average: 86.08 High Score: 99 Low Score: 35
    Exam #2 Average: 86.42 High Score: 94 Low Score: 72
    Exam #3 Average: 84.75 High Score: 91 Low Score: 65
    Exam #4 Average: 76.92 High Score: 94 Low Score: 39
 */
```

La consulta funciona usando el método Split para convertir cada línea de texto en una matriz. Cada elemento de matriz representa una columna. Por último, el texto de cada columna se convierte en su representación numérica. Si el archivo es un archivo separado por tabulaciones, actualice el argumento del método `Split` a `\t`.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ y cadenas (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para consultar los metadatos de un ensamblado con reflexión (LINQ) (C#)

29/06/2020 • 2 minutes to read • Edit Online

Las API de reflexión de .NET Framework se pueden usar para examinar los metadatos de un ensamblado .NET y para crear colecciones de tipos, escribir miembros, parámetros y otros elementos que se encuentren en ese ensamblado. Dado que estas colecciones admiten la interfaz genérica IEnumerable<T>, se pueden consultar mediante LINQ.

En el ejemplo siguiente se muestra cómo se puede usar LINQ con reflexión para recuperar metadatos concretos sobre métodos que coinciden con un criterio de búsqueda especificado. En este caso, la consulta encontrará los nombres de todos los métodos del ensamblado que devuelven tipos enumerables, como matrices.

Ejemplo

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                        from method in type.GetMethods()
                        where method.ReturnType.IsArray == true
                            || ( method.ReturnType.GetInterface(
                                typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                            && method.ReturnType.FullName != "System.String" )
                        group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine("  {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

El ejemplo usa el método Assembly.GetTypes para devolver una matriz de tipos en el ensamblado especificado. Se aplica el filtro where para que solo se devuelvan tipos públicos. Para cada tipo público, se genera una consulta anidada con la matriz MethodInfo que se devuelve desde la llamada Type.GetMethods. Estos resultados se filtran para que solo devuelvan los métodos cuyo tipo de valor devuelto sea una matriz, o un tipo que implemente IEnumerable<T>. Por último, estos resultados se agrupan usando el nombre de tipo como una clave.

Vea también

- LINQ to Objects (C#)

Procedimiento para consultar los metadatos de un ensamblado con reflexión (LINQ) (C#)

29/06/2020 • 2 minutes to read • Edit Online

Las API de reflexión de .NET Framework se pueden usar para examinar los metadatos de un ensamblado .NET y para crear colecciones de tipos, escribir miembros, parámetros y otros elementos que se encuentren en ese ensamblado. Dado que estas colecciones admiten la interfaz genérica IEnumerable<T>, se pueden consultar mediante LINQ.

En el ejemplo siguiente se muestra cómo se puede usar LINQ con reflexión para recuperar metadatos concretos sobre métodos que coinciden con un criterio de búsqueda especificado. En este caso, la consulta encontrará los nombres de todos los métodos del ensamblado que devuelven tipos enumerables, como matrices.

Ejemplo

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089");
        var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                        from method in type.GetMethods()
                        where method.ReturnType.IsArray == true
                            || ( method.ReturnType.GetInterface(
                                typeof(System.Collections.Generic.IEnumerable<>).FullName ) != null
                            && method.ReturnType.FullName != "System.String" )
                        group method.ToString() by type.ToString();

        foreach (var groupOfMethods in pubTypesQuery)
        {
            Console.WriteLine("Type: {0}", groupOfMethods.Key);
            foreach (var method in groupOfMethods)
            {
                Console.WriteLine("  {0}", method);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

El ejemplo usa el método Assembly.GetTypes para devolver una matriz de tipos en el ensamblado especificado. Se aplica el filtro where para que solo se devuelvan tipos públicos. Para cada tipo público, se genera una consulta anidada con la matriz MethodInfo que se devuelve desde la llamada Type.GetMethods. Estos resultados se filtran para que solo devuelvan los métodos cuyo tipo de valor devuelto sea una matriz, o un tipo que implemente IEnumerable<T>. Por último, estos resultados se agrupan usando el nombre de tipo como una clave.

Vea también

- LINQ to Objects (C#)

LINQ y directorios de archivos (C#)

18/03/2020 • 4 minutes to read • Edit Online

Muchas operaciones de sistema de archivos son esencialmente consultas y, por tanto, son adecuadas para el enfoque LINQ.

Las consultas en esta sección no son destructivas. No se usan para cambiar el contenido de las carpetas o los archivos originales. Esto sigue la regla de que las consultas no deben causar efectos secundarios. En general, cualquier código (incluidas las consultas que ejecutan operadores de creación actualización y eliminación) que modifica los datos de origen se debe separar del código que solo consulta los datos.

Esta sección contiene los siguientes temas:

Búsqueda de archivos con un nombre o atributo especificados (C#)
Muestra cómo buscar archivos examinando una o más propiedades de su objeto FileInfo.

Agrupación de archivos por extensión (LINQ) (C#)
Muestra cómo devolver grupos del objeto FileInfo basándose en su extensión de nombre de archivo.

Búsqueda del número total de bytes en un conjunto de carpetas (LINQ) (C#)
Muestra cómo devolver el número total de bytes en todos los archivos en un árbol de directorio especificado.

Procedimiento para comparar el contenido de dos carpetas (LINQ) (C#)
Muestra cómo devolver todos los archivos que se encuentran en dos carpetas especificadas y también todos los archivos que se encuentran en una carpeta pero no en la otra.

Búsqueda del archivo o archivos de mayor tamaño en un árbol de directorios (LINQ) (C#)
Muestra cómo devolver el archivo mayor o menor, o un número especificado de archivos, en un árbol de directorios.

Búsqueda de archivos duplicados en un árbol de directorios (LINQ) (C#)
Muestra cómo agrupar todos los nombres de archivo que aparecen en más de una ubicación en un árbol de directorio especificado. También muestra cómo realizar comparaciones más complejas basadas en un comparador personalizado.

Consulta del contenido de los archivos de una carpeta (LINQ) (C#)
Muestra cómo recorrer en iteración las carpetas de un árbol, abrir cada archivo y consultar el contenido del archivo.

Comentarios

Hay cierta complejidad en la creación de un origen de datos que representa de forma precisa el contenido del sistema de archivos y controla las excepciones correctamente. En los ejemplos de esta sección se crea una colección de instantáneas de objetos FileInfo que representa todos los archivos en una carpeta raíz especificada y todas sus subcarpetas. El estado real de cada FileInfo puede cambiar en el periodo comprendido entre el comienzo y el fin de la ejecución de una consulta. Por ejemplo, se puede crear una lista de objetos FileInfo para usarla como origen de datos. Si se intenta tener acceso a la propiedad `Length` en una consulta, el objeto FileInfo intentará tener acceso al sistema de archivos para actualizar el valor de `Length`. Si el archivo ya no existe, se obtendrá una excepción FileNotFoundException en la consulta, aunque no se esté consultando el sistema de archivos directamente. Algunas consultas de esta sección usan un método independiente que consume estas excepciones concretas en casos determinados. Otra opción consiste en mantener actualizado el origen de datos de manera dinámica mediante FileSystemWatcher.

Vea también

- LINQ to Objects (C#)

Procedimiento para buscar archivos con un nombre o atributo especificados (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo encontrar todos los archivos con una determinada extensión de nombre de archivo (por ejemplo, ".txt") en un árbol de directorios especificado. También se muestra cómo devolver el archivo más reciente o más antiguo del árbol por fecha de creación.

Ejemplo

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
             orderby file.CreationTime
             select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ to Objects (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para agrupar archivos por extensión (LINQ) (C#)

18/03/2020 • 4 minutes to read • Edit Online

En este ejemplo se muestra cómo se puede usar LINQ para efectuar operaciones avanzadas de agrupación y ordenación en listas de archivos o de carpetas. También muestra cómo paginar la salida en la ventana de consola mediante los métodos Skip y Take.

Ejemplo

En la siguiente consulta se muestra cómo agrupar el contenido de un árbol de directorio especificado por la extensión de nombre de archivo.

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    //  and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects with string keys.
    // It can be modified to work for any long listings of data. Note that explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
                                   IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;
```

```
        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;

            // Output only as many lines of the current group as will fit in the window.
            do
            {
                Console.Clear();
                Console.WriteLine(filegroup.Key == String.Empty ? "[none]" : filegroup.Key);

                // Get 'numLines' number of items starting at number 'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);

                //Execute the resultPage query
                foreach (var f in resultPage)
                {
                    Console.WriteLine("\t{0}", f.FullName.Substring(rootLength));
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End' key to break...");
                ConsoleKey key = Console.ReadKey().Key;
                if (key == ConsoleKey.End)
                {
                    goAgain = false;
                    break;
                }
            } while (currentLine < filegroup.Count());

            if (goAgain == false)
                break;
        }
    }
}
```

La salida de este programa puede ser larga, dependiendo de los detalles del sistema de archivos local y de la configuración de `startFolder`. Para habilitar la visualización de todos los resultados, en este ejemplo se muestra cómo paginar los resultados. Se pueden aplicar las mismas técnicas a las aplicaciones web y Windows. Observe que, como el código pagina los elementos en un grupo, se necesita un bucle `foreach` anidado. También hay alguna lógica adicional para calcular la posición actual en la lista y para permitir que el usuario detenga la paginación y salga del programa. En este caso en concreto, la consulta de paginación se ejecuta en los resultados almacenados en caché de la consulta original. En otros contextos, como en LINQ to SQL, este almacenamiento en caché no es necesario.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ to Objects (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para buscar el número total de bytes de un conjunto de carpetas (LINQ) (C#)

18/03/2020 • 4 minutes to read • Edit Online

En este ejemplo se muestra cómo recuperar el número total de bytes usados por todos los archivos en una carpeta especificada y en todas sus subcarpetas.

Ejemplo

El método Sum agrega los valores de todos los elementos seleccionados en la cláusula `select`. Puede modificar fácilmente esta consulta para recuperar el archivo más grande y más pequeño en el árbol de directorio especificado llamando al método Min o Max en lugar de Sum.

```
class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList = System.IO.Directory.GetFiles(startFolder, "*.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // Cache the results to avoid multiple trips to the file system.
        long[] fileLengths = fileQuery.ToArray();

        // Return the size of the largest file
        long largestFile = fileLengths.Max();

        // Return the total number of bytes in all the files under the specified folder.
        long totalBytes = fileLengths.Sum();

        Console.WriteLine("There are {0} bytes in {1} files under {2}",
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}
```

Si solo tiene que contar el número de bytes en un árbol de directorio especificado, puede hacerlo de forma más eficaz sin crear una consulta LINQ, ya que conlleva la sobrecarga de crear la colección de listas como un origen de datos. La utilidad del enfoque de LINQ aumenta a medida que la consulta se vuelve más compleja, o cuando se tienen que ejecutar varias consultas en el mismo origen de datos.

La consulta llama a un método independiente para obtener la longitud del archivo. Lo hace para consumir la excepción que probablemente se producirá si el archivo se ha eliminado en otro subproceso después de que se creara el objeto FileInfo en la llamada a `GetFiles`. Aunque ya se haya creado el objeto FileInfo, puede producirse una excepción porque un objeto FileInfo intentará actualizar su propiedad Length con la longitud más actual la primera vez que se tenga acceso a la propiedad. Al incluir esta operación en un bloque try-catch fuera de la consulta, el código sigue la regla de evitar las operaciones en las consultas que pueden producir efectos

secundarios. En general, debe tener mucho cuidado al consumir excepciones para asegurarse de que no deja una aplicación en un estado desconocido.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ to Objects (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para comparar el contenido de dos carpetas (LINQ) (C#)

18/03/2020 • 3 minutes to read • Edit Online

En este ejemplo se muestran tres maneras de comparar dos listados de archivos:

- Mediante la consulta de un valor booleano que especifica si las dos listas de archivos son idénticas.
- Mediante la consulta de la intersección para recuperar los archivos que están en ambas carpetas.
- Mediante la consulta de la diferencia de conjuntos para recuperar los archivos que se encuentran en una carpeta, pero no en la otra.

> **NOTE**
>
> Las técnicas que se mencionan aquí pueden adaptarse para comparar secuencias de objetos de cualquier tipo.

La clase `FileComparer` que aparece a continuación muestra cómo usar una clase de comparador personalizada junto con los operadores de consulta estándar. La clase no está diseñada para su uso en escenarios reales. Simplemente usa el nombre y la longitud en bytes de cada archivo para determinar si el contenido de cada una de las carpetas es idéntico o no. En un escenario real, debería modificar este comparador para realizar una comprobación de igualdad más rigurosa.

Ejemplo

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {

        static void Main(string[] args)
        {

            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";

            System.IO.DirectoryInfo dir1 = new System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

            //A custom file comparer defined below
            FileCompare myFileCompare = new FileCompare();

            // This query determines whether the two folders contain
            // identical file lists, based on the custom file comparer
            // that is defined in the FileCompare class.
            // The query executes immediately because it returns a bool.
            bool areIdentical = list1.SequenceEqual(list2, myFileCompare);
```

```
            if (areIdentical == true)
            {
                Console.WriteLine("the two folders are the same");
            }
            else
            {
                Console.WriteLine("The two folders are not the same");
            }

            // Find the common files. It produces a sequence and doesn't
            // execute until the foreach statement.
            var queryCommonFiles = list1.Intersect(list2, myFileCompare);

            if (queryCommonFiles.Any())
            {
                Console.WriteLine("The following files are in both folders:");
                foreach (var v in queryCommonFiles)
                {
                    Console.WriteLine(v.FullName); //shows which items end up in result list
                }
            }
            else
            {
                Console.WriteLine("There are no common files in the two folders.");
            }

            // Find the set difference between the two folders.
            // For this example we only check one way.
            var queryList1Only = (from file in list1
                                  select file).Except(list2, myFileCompare);

            Console.WriteLine("The following files are in list1 but not list2:");
            foreach (var v in queryList1Only)
            {
                Console.WriteLine(v.FullName);
            }

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    // This implementation defines a very simple comparison
    // between two FileInfo objects. It only compares the name
    // of the files being compared and their length in bytes.
    class FileCompare : System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
    {
        public FileCompare() { }

        public bool Equals(System.IO.FileInfo f1, System.IO.FileInfo f2)
        {
            return (f1.Name == f2.Name &&
                    f1.Length == f2.Length);
        }

        // Return a hash that reflects the comparison criteria. According to the
        // rules for IEqualityComparer<T>, if Equals is true, then the hash codes must
        // also be equal. Because equality as defined here is a simple value equality, not
        // reference identity, it is possible that two or more objects will produce the same
        // hash code.
        public int GetHashCode(System.IO.FileInfo fi)
        {
            string s = $"{fi.Name}{fi.Length}";
            return s.GetHashCode();
        }
    }
}
```

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ to Objects (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para buscar el archivo o archivos de mayor tamaño en un árbol de directorios (LINQ) (C#)

18/03/2020 • 5 minutes to read • Edit Online

En este ejemplo se muestran cinco consultas relacionadas con el tamaño de archivo en bytes:

- Cómo recuperar el tamaño en bytes del archivo más grande.
- Cómo recuperar el tamaño en bytes del archivo más pequeño.
- Cómo recuperar el archivo de mayor o menor tamaño del objeto FileInfo de una o más carpetas en una carpeta raíz especificada.
- Cómo recuperar una secuencia, como los 10 archivos de mayor tamaño.
- Cómo ordenar los archivos en grupos según su tamaño en bytes, sin incluir los archivos inferiores a un tamaño especificado.

Ejemplo

El ejemplo siguiente contiene cinco consultas independientes que muestran cómo consultar y agrupar archivos, en función de su tamaño en bytes. Puede modificar fácilmente estos ejemplos para basar la consulta en otra propiedad del objeto FileInfo.

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
             let len = GetFileLength(file)
             select len)
             .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
            startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
```

```
        System.IO.FileInfo longestFile =
            (from file in fileList
             let len = GetFileLength(file)
             where len > 0
             orderby len descending
             select file)
            .First();

        Console.WriteLine("The largest file under {0} is {1} with a length of {2} bytes",
                          startFolder, longestFile.FullName, longestFile.Length);

        //Return the FileInfo of the smallest file
        System.IO.FileInfo smallestFile =
            (from file in fileList
             let len = GetFileLength(file)
             where len > 0
             orderby len ascending
             select file).First();

        Console.WriteLine("The smallest file under {0} is {1} with a length of {2} bytes",
                          startFolder, smallestFile.FullName, smallestFile.Length);

        //Return the FileInfos for the 10 largest files
        // queryTenLargest is an IEnumerable<System.IO.FileInfo>
        var queryTenLargest =
            (from file in fileList
             let len = GetFileLength(file)
             orderby len descending
             select file).Take(10);

        Console.WriteLine("The 10 largest files under {0} are:", startFolder);

        foreach (var v in queryTenLargest)
        {
            Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
        }

        // Group the files according to their size, leaving out
        // files that are less than 200000 bytes.
        var querySizeGroups =
            from file in fileList
            let len = GetFileLength(file)
            where len > 0
            group file by (len / 100000) into fileGroup
            where fileGroup.Key >= 2
            orderby fileGroup.Key descending
            select fileGroup;

        foreach (var filegroup in querySizeGroups)
        {
            Console.WriteLine(filegroup.Key.ToString() + "00000");
            foreach (var item in filegroup)
            {
                Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
            }
        }
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the FileInfo.Length property.
    // In this particular case, it is safe to swallow the exception.
    static long GetFileLength(System.IO.FileInfo fi)
    {
        long retval;
        try
        {
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
```

```
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }

}
```

Para devolver uno o más objetos FileInfo completos, la consulta debe examinar cada uno de ellos en los datos de origen y, después, ordenarlos por el valor de su propiedad Length. Después, puede devolver el objeto único o la secuencia con la mayor longitud. Use First para devolver el primer elemento de una lista. Use Take para devolver el primer número n de elementos. Especifique un criterio de ordenación descendente para colocar los elementos más pequeños al principio de la lista.

La consulta llama a un método independiente para obtener el tamaño del archivo en bytes para consumir la posible excepción que se generará en el caso de que se haya eliminado un archivo en otro subproceso en el período de tiempo desde que se ha creado el objeto FileInfo en la llamada a `GetFiles`. Aunque ya se haya creado el objeto FileInfo, puede producirse una excepción porque un objeto FileInfo intentará actualizar su propiedad Length con el tamaño más actual la primera vez que se tenga acceso a la propiedad. Al incluir esta operación en un bloque try-catch fuera de la consulta, seguimos la regla de evitar las operaciones en las consultas que pueden producir efectos secundarios. En general, debe tener mucho cuidado al consumir excepciones para asegurarse de que no deja una aplicación en un estado desconocido.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ to Objects (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para buscar archivos duplicados en un árbol de directorios (LINQ) (C#)

18/03/2020 • 4 minutes to read • Edit Online

A veces, los archivos que tienen el mismo nombre pueden estar en más de una carpeta. Por ejemplo, en la carpeta de instalación de Visual Studio, hay varias carpetas que tienen un archivo readme.htm. En este ejemplo se muestra cómo buscar estos nombres de archivos duplicados en una carpeta raíz especificada. En el segundo ejemplo se muestra cómo buscar archivos cuyo tamaño y fecha de LastWrite también coinciden.

Ejemplo

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        // used in WriteLine to keep the lines shorter
        int charsToSkip = startFolder.Length;

        // var can be used for convenience with groups.
        var queryDupNames =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by file.Name into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
        public long Length { get; set; }
```

```
        public override bool Equals(object obj)
        {
            PortableKey other = (PortableKey)obj;
            return other.LastWriteTime == this.LastWriteTime &&
                   other.Length == this.Length &&
                   other.Name == this.Name;
        }

        public override int GetHashCode()
        {
            string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
            return str.GetHashCode();
        }
        public override string ToString()
        {
            return $"{this.Name} {this.Length} {this.LastWriteTime}";
        }
    }
    static void QueryDuplicates2()
    {
        // Change the root drive or folder if necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Make the lines shorter for the console display
        int charsToSkip = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        // Note the use of a compound key. Files that match
        // all three properties belong to the same group.
        // A named type is used to enable the query to be
        // passed to another method. Anonymous types can also be used
        // for composite keys but cannot be passed across method boundaries
        //
        var queryDupFiles =
            from file in fileList
            group file.FullName.Substring(charsToSkip) by
                new PortableKey { Name = file.Name, LastWriteTime = file.LastWriteTime, Length = file.Length }
into fileGroup
            where fileGroup.Count() > 1
            select fileGroup;

        var list = queryDupFiles.ToList();

        int i = queryDupFiles.Count();

        PageOutput<PortableKey, string>(queryDupFiles);
    }

    // A generic method to page the output of the QueryDuplications methods
    // Here the type of the group must be specified explicitly. "var" cannot
    // be used in method signatures. This method does not display more than one
    // group per page.
    private static void PageOutput<K, V>(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            int currentLine = 0;
```

```
            // Output only as many lines of the current group as will fit in the window.
            do
            {
                Console.Clear();
                Console.WriteLine("Filename = {0}", filegroup.Key.ToString() == String.Empty ? "[none]" :
filegroup.Key.ToString());

                // Get 'numLines' number of items starting at number 'currentLine'.
                var resultPage = filegroup.Skip(currentLine).Take(numLines);

                //Execute the resultPage query
                foreach (var fileName in resultPage)
                {
                    Console.WriteLine("\t{0}", fileName);
                }

                // Increment the line counter.
                currentLine += numLines;

                // Give the user a chance to escape.
                Console.WriteLine("Press any key to continue or the 'End' key to break...");
                ConsoleKey key = Console.ReadKey().Key;
                if (key == ConsoleKey.End)
                {
                    goAgain = false;
                    break;
                }
            } while (currentLine < filegroup.Count());

            if (goAgain == false)
                break;
        }
    }
}
```

En la primera consulta se usa una clave simple para determinar una coincidencia; se buscan archivos que tengan el mismo nombre, pero cuyo contenido podría ser diferente. En la segunda consulta se usa una clave compuesta para coincidir con tres propiedades del objeto FileInfo. En esta consulta es mucho más probable que se encuentren archivos que tienen el mismo nombre y un contenido similar o idéntico.

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ to Objects (C#)
- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])

Procedimiento para consultar el contenido de los archivos de texto de una carpeta (LINQ) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo consultar todos los archivos en un árbol de directorios especificado, abrir cada archivo e inspeccionar su contenido. Este tipo de técnica puede usarse para crear índices o índices inversos del contenido de un árbol de directorios. En este ejemplo, se realiza una búsqueda de cadena simple. Pero los tipos más complejos de coincidencia de patrones se pueden realizar con una expresión regular. Para obtener más información, vea Procedimiento para combinar consultas LINQ con expresiones regulares (C#).

Ejemplo

```
class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*", System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the RegEx class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}
```

Compilar el código

Cree un proyecto de aplicación de consola de C# con directivas `using` para los espacios de nombres System.Linq y System.IO.

Vea también

- LINQ and File Directories (C#) (LINQ y directorios de archivos [C#])
- LINQ to Objects (C#)

Procedimiento para consultar un objeto ArrayList con LINQ (C#)

18/03/2020 • 2 minutes to read • Edit Online

Cuando use LINQ para consultar colecciones no genéricas IEnumerable como ArrayList, debe declarar explícitamente el tipo de variable de rango para reflejar el tipo específico de los objetos de la colección. Por ejemplo, si tiene una ArrayList de objetos `Student`, la cláusula from debe tener un aspecto similar a este:

```
var query = from Student s in arrList
//...
```

Al especificar el tipo de la variable de rango, se convierte cada elemento de la ArrayList en un `Student`.

El uso de una variable de rango con tipo explícito en una expresión de consulta es equivalente a llamar al método Cast. Cast genera una excepción si la conversión especificada no puede realizarse. Cast y OfType son los dos métodos de operador de consulta estándar que funcionan en tipos IEnumerable no genéricos. Para obtener más información, vea Relaciones entre tipos en operaciones de consulta LINQ.

Ejemplo

En el siguiente ejemplo se muestra una consulta simple sobre un ArrayList. Tenga en cuenta que en este ejemplo se usan inicializadores de objeto cuando el código llama al método Add, pero esto no es un requisito.

```
using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                    {
                        FirstName = "Svetlana", LastName = "Omelchenko", Scores = new int[] { 98, 92, 81, 60
}
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Sven", LastName = "Mortensen", Scores = new int[] { 88, 94, 65, 91 }
                    });
            arrList.Add(
                new Student
                    {
                        FirstName = "Cesar", LastName = "Garcia", Scores = new int[] { 97, 89, 85, 82 }
                    });

            var query = from Student student in arrList
                        where student.Scores[0] > 95
                        select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
/* Output:
    Omelchenko: 98
    Garcia: 97
*/
```

Vea también

- LINQ to Objects (C#)

Procedimiento para agregar métodos personalizados para las consultas LINQ (C#)

29/06/2020 • 8 minutes to read • Edit Online

Puede extender el conjunto de métodos que puede usar para consultas LINQ agregando métodos de extensión a la interfaz IEnumerable<T>. Por ejemplo, además de las operaciones habituales de promedio o de máximo, puede crear un método de agregación personalizado para calcular un valor a partir de una secuencia de valores. También puede crear un método que funcione como un filtro personalizado o como una transformación de datos específica para una secuencia de valores y que devuelva una secuencia nueva. Ejemplos de dichos métodos son Distinct, Skip y Reverse.

Si extiende la interfaz IEnumerable<T>, puede aplicar los métodos personalizados a cualquier colección enumerable. Para más información, vea Métodos de extensión.

Agregar un método de agregación

Un método de agregación calcula un valor único a partir de un conjunto de valores. LINQ proporciona varios métodos de agregación, incluidos Average, Min y Max. Si quiere crear su propio método de agregación, agregue un método de extensión a la interfaz IEnumerable<T>.

En el ejemplo de código siguiente se muestra cómo crear un método de extensión denominado `Median` para calcular la mediana de una secuencia de números de tipo `double`.

```
public static class LINQExtension
{
    public static double Median(this IEnumerable<double> source)
    {
        var countOfElementsInTheSet = source?.Count() ?? 0;

        if (countOfElementsInTheSet == 0)
        {
            throw new InvalidOperationException("Cannot compute median for a null or empty set.");
        }

        var sortedList = (from number in source
                          orderby number
                          select number).ToList();

        int itemIndex = countOfElementsInTheSet / 2;

        if (countOfElementsInTheSet % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Puede llamar a este método de extensión para cualquier colección enumerable de la misma manera en la que llamaría a otros métodos de agregación desde la interfaz IEnumerable<T>.

En el ejemplo de código siguiente se muestra cómo usar el método `Median` para una matriz de tipo `double`.

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };

var query1 = numbers1.Median();

Console.WriteLine("double: Median = " + query1);
```

```
/*
 This code produces the following output:

 Double: Median = 4.85
*/
```

Sobrecargar un método de agregación para que acepte varios tipos

Puede sobrecargar el método de agregación para que acepte secuencias de varios tipos. El enfoque estándar consiste en crear una sobrecarga para cada tipo. Otro enfoque consiste en crear una sobrecarga que tome un tipo genérico y lo convierta a un tipo específico mediante un delegado. También puede combinar ambos enfoques.

Para crear una sobrecarga para cada tipo

Puede crear una sobrecarga específica para cada tipo que desee admitir. En el siguiente ejemplo de código se muestra una sobrecarga del método `Median` para el tipo `integer`.

```
//int overload

public static double Median(this IEnumerable<int> source)
{
    return (from num in source select (double)num).Median();
}
```

Ahora puede llamar a las sobrecargas `Median` para los tipos `integer` y `double`, como se muestra en el código siguiente:

```
double[] numbers1 = { 1.9, 2, 8, 4, 5.7, 6, 7.2, 0 };

var query1 = numbers1.Median();

Console.WriteLine("double: Median = " + query1);
```

```
int[] numbers2 = { 1, 2, 3, 4, 5 };

var query2 = numbers2.Median();

Console.WriteLine("int: Median = " + query2);
```

```
/*
 This code produces the following output:

 Double: Median = 4.85
 Integer: Median = 3
*/
```

Para crear una sobrecarga genérica

También puede crear una sobrecarga que acepte una secuencia de objetos genéricos. Esta sobrecarga toma un delegado como parámetro y lo usa para convertir una secuencia de objetos de un tipo genérico a un tipo

específico.

En el código siguiente se muestra una sobrecarga del método `Median` que toma el delegado Func<T,TResult> como parámetro. Este delegado toma un objeto del tipo genérico T y devuelve un objeto de tipo `double`.

```
// Generic overload.

public static double Median<T>(this IEnumerable<T> numbers,
                       Func<T, double> selector)
{
    return (from num in numbers select selector(num)).Median();
}
```

Ahora puede llamar al método `Median` para una secuencia de objetos de cualquier tipo. Si el tipo no tiene su propia sobrecarga de métodos, deberá pasar un parámetro de delegado. En C# puede usar una expresión lambda para este propósito. Además, solo en Visual Basic, si usa la cláusula `Aggregate` o `Group By` en lugar de la llamada al método, puede pasar cualquier valor o expresión que esté en el ámbito de esta cláusula.

En el ejemplo de código siguiente se muestra cómo llamar al método `Median` para una matriz de enteros y una matriz de cadenas. Para las cadenas, se calcula la mediana de las longitudes de las cadenas de la matriz. En el ejemplo se muestra cómo pasar el parámetro del delegado Func<T,TResult> al método `Median` para cada caso.

```
int[] numbers3 = { 1, 2, 3, 4, 5 };

/*
  You can use the num=>num lambda expression as a parameter for the Median method
  so that the compiler will implicitly convert its value to double.
  If there is no implicit conversion, the compiler will display an error message.
*/

var query3 = numbers3.Median(num => num);

Console.WriteLine("int: Median = " + query3);

string[] numbers4 = { "one", "two", "three", "four", "five" };

// With the generic overload, you can also use numeric properties of objects.

var query4 = numbers4.Median(str => str.Length);

Console.WriteLine("String: Median = " + query4);

/*
 This code produces the following output:

 Integer: Median = 3
 String: Median = 4
*/
```

Agregar un método que devuelva una colección

Puede extender la interfaz IEnumerable<T> con un método de consulta personalizado que devuelve una secuencia de valores. En este caso, el método debe devolver una colección de tipo IEnumerable<T>. Estos métodos se pueden usar para aplicar filtros o transformaciones de datos a una secuencia de valores.

En el ejemplo siguiente se muestra cómo crear un método de extensión denominado `AlternateElements` que devuelve los demás elementos de una colección, empezando por el primer elemento.

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.

public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    List<T> list = new List<T>();

    int i = 0;

    foreach (var element in source)
    {
        if (i % 2 == 0)
        {
            list.Add(element);
        }

        i++;
    }

    return list;
}
```

Puede llamar a este método de extensión para cualquier colección enumerable de la misma manera en la que llamaría a otros métodos desde la interfaz IEnumerable<T>, como se muestra en el siguiente código:

```
string[] strings = { "a", "b", "c", "d", "e" };

var query = strings.AlternateElements();

foreach (var element in query)
{
    Console.WriteLine(element);
}
/*
 This code produces the following output:

 a
 c
 e
*/
```

Vea también

- IEnumerable<T>
- Métodos de extensión

Información general de LINQ to XML (C#)

29/06/2020 • 8 minutes to read • Edit Online

LINQ to XML proporciona una interfaz de programación XML en memoria que aprovecha .NET Language-Integrated Query (LINQ) Framework. LINQ to XML utiliza las características de .NET y es comparable a una actualizada y rediseñada interfaz de programación XML para el Modelo de objetos de documento (DOM).

XML se ha adoptado ampliamente como un modo de dar formato a datos en diversos contextos. Por ejemplo, puede encontrar XML en la web, en archivos de configuración, en archivos de Microsoft Office Word y en bases de datos.

LINQ to XML es un método actualizado y rediseñado para la programación con XML. Proporciona las funcionalidades de modificación de documentos en memoria del Modelo de objetos de documento (DOM) y es compatible con expresiones de consulta LINQ. Aunque estas expresiones de consulta difieren sintácticamente de XPath, proporcionan una funcionalidad similar.

Desarrolladores de LINQ to XML

LINQ to XML se dirige a diversos desarrolladores. Para el desarrollador medio que solo desea completar una tarea, LINQ to XML simplifica el código XML al proporcionar una experiencia de consulta similar a SQL. Con un poco de estudio, los programadores pueden aprender a escribir consultas sucintas y eficaces en el lenguaje de programación que prefieran.

Los desarrolladores profesionales pueden usar LINQ to XML para aumentar considerablemente su productividad. Con LINQ to XML, pueden escribir menos código, que a su vez resulte más expresivo, compacto y eficaz. Pueden usar expresiones de consulta de distintos dominios de datos simultáneamente.

¿Qué es LINQ to XML?

LINQ to XML es una interfaz de programación XML en memoria y habilitada para LINQ que permite trabajar con XML desde los lenguajes de programación de .NET.

LINQ to XML se parece a Document Object Model (DOM) en lo que respecta a la inserción del documento XML en la memoria. Puede consultar y modificar el documento; una vez modificado, puede guardarlo en un archivo o serializarlo y enviarlo a través de Internet. Pero LINQ to XML es diferente de DOM: proporciona un nuevo modelo de objetos más ligero, con el que se trabaja más fácilmente y que aprovecha las características de lenguaje de C#.

La ventaja más importante de LINQ to XML es su integración con Language-Integrated Query (LINQ). Esta integración permite escribir consultas en el documento XML en memoria para recuperar colecciones de elementos y atributos. La capacidad de consulta de LINQ to XML es comparable en cuanto a funcionalidad (aunque no cuanto a sintaxis) a XPath y XQuery. La integración de LINK en C# proporciona una escritura más rápida, comprobación en tiempo de compilación y compatibilidad mejorada con el depurador.

Otra ventaja de LINQ to XML es la capacidad de usar los resultados de la consulta como parámetros en constructores de objetos XElement y XAttribute, que habilita un método eficaz para crear árboles XML. Este método, denominado *construcción funcional*, permite que los desarrolladores transformen fácilmente árboles XML de una forma a otra.

Por ejemplo, es posible que tenga un pedido de compra XML típico, como se describe en Archivo XML de ejemplo: Pedido de compra común (LINQ to XML). Mediante LINQ to XML, podría ejecutar la siguiente consulta para obtener el valor del atributo de número de pieza relativo a cada elemento del pedido de compra:

```
// Load the XML file from our project directory containing the purchase orders
var filename = "PurchaseOrder.xml";
var currentDirectory = Directory.GetCurrentDirectory();
var purchaseOrderFilepath = Path.Combine(currentDirectory, filename);

XElement purchaseOrder = XElement.Load(purchaseOrderFilepath);

IEnumerable<string> partNos =  from item in purchaseOrder.Descendants("Item")
                               select (string) item.Attribute("PartNumber");
```

Esto se puede reescribir con formato de sintaxis de método:

```
IEnumerable<string> partNos = purchaseOrder.Descendants("Item").Select(x => (string)
x.Attribute("PartNumber"));
```

A modo de ejemplo, imagine que necesita una lista, ordenada por números de pieza, de los elementos con un valor superior a 100 $. Para obtener esta información, podría ejecutar la siguiente consulta:

```
// Load the XML file from our project directory containing the purchase orders
var filename = "PurchaseOrder.xml";
var currentDirectory = Directory.GetCurrentDirectory();
var purchaseOrderFilepath = Path.Combine(currentDirectory, filename);

XElement purchaseOrder = XElement.Load(purchaseOrderFilepath);

IEnumerable<XElement> pricesByPartNos =  from item in purchaseOrder.Descendants("Item")
                                 where (int) item.Element("Quantity") * (decimal) item.Element("USPrice") >
100
                                 orderby (string)item.Element("PartNumber")
                                 select item;
```

De nuevo, esto se puede reescribir con formato de sintaxis de método:

```
IEnumerable<XElement> pricesByPartNos = purchaseOrder.Descendants("Item")
                                        .Where(item => (int)item.Element("Quantity") *
(decimal)item.Element("USPrice") > 100)
                                        .OrderBy(order => order.Element("PartNumber"));
```

Además de estas funcionalidades de LINK, LINQ to XML proporciona una interfaz de programación XML mejorada. Con LINQ to XML, se puede:

- Cargar XML a partir de archivos o secuencias.
- Serializar XML a archivos o secuencias.
- Crear árboles XML desde cero mediante la construcción funcional.
- Realizar consultas de XML con ejes de tipo XPath.
- Manipular el árbol XML en memoria con métodos como Add, Remove, ReplaceWith y SetValue.
- Validar árboles XML mediante XSD.
- Usar una combinación de estas características para transformar las formas de los árboles XML.

Crear árboles XML

Una de las ventajas más significativas de la programación con LINQ to XML es la facilidad con que se pueden crear árboles XML. Por ejemplo, para crear un árbol XML pequeño, puede escribir código de la siguiente

manera:

```
XElement contacts =
new XElement("Contacts",
    new XElement("Contact",
        new XElement("Name", "Patrick Hines"),
        new XElement("Phone", "206-555-0144",
            new XAttribute("Type", "Home")),
        new XElement("phone", "425-555-0145",
            new XAttribute("Type", "Work")),
        new XElement("Address",
            new XElement("Street1", "123 Main St"),
            new XElement("City", "Mercer Island"),
            new XElement("State", "WA"),
            new XElement("Postal", "68042")
        )
    )
);
```

Para obtener más información, consulte Crear árboles XML (C#).

Vea también

- Referencia (LINQ to XML)
- LINQ to XML frente a DOM (C#)
- LINQ to XML frente a otras tecnologías XML
- System.Xml.Linq

LINQ to XML frente a DOM (C#)

18/03/2020 • 11 minutes to read • Edit Online

En esta sección se describen algunas diferencias fundamentales entre LINQ to XML y la API de programación XML predominante actual, Document Object Model (DOM) W3C.

Nuevas formas de crear árboles XML

En DOM W3C, los árboles XML se crean de abajo arriba; es decir, se crea un documento, se crean elementos y, a continuación, se agregan los elementos al documento.

Por ejemplo, a continuación se muestra una forma típica de crear un árbol XML usando la implementación de DOM de Microsoft, XmlDocument:

```
XmlDocument doc = new XmlDocument();
XmlElement name = doc.CreateElement("Name");
name.InnerText = "Patrick Hines";
XmlElement phone1 = doc.CreateElement("Phone");
phone1.SetAttribute("Type", "Home");
phone1.InnerText = "206-555-0144";
XmlElement phone2 = doc.CreateElement("Phone");
phone2.SetAttribute("Type", "Work");
phone2.InnerText = "425-555-0145";
XmlElement street1 = doc.CreateElement("Street1");
street1.InnerText = "123 Main St";
XmlElement city = doc.CreateElement("City");
city.InnerText = "Mercer Island";
XmlElement state = doc.CreateElement("State");
state.InnerText = "WA";
XmlElement postal = doc.CreateElement("Postal");
postal.InnerText = "68042";
XmlElement address = doc.CreateElement("Address");
address.AppendChild(street1);
address.AppendChild(city);
address.AppendChild(state);
address.AppendChild(postal);
XmlElement contact = doc.CreateElement("Contact");
contact.AppendChild(name);
contact.AppendChild(phone1);
contact.AppendChild(phone2);
contact.AppendChild(address);
XmlElement contacts = doc.CreateElement("Contacts");
contacts.AppendChild(contact);
doc.AppendChild(contacts);
```

Este estilo de codificación no proporciona mucha información visual acerca de la estructura del árbol XML. LINQ to XML admite este enfoque para crear un árbol XML, pero también admite un enfoque alternativo, la *construcción funcional*. La construcción funcional usa los constructores XElement y XAttribute para crear un árbol XML.

A continuación se muestra cómo se crea el mismo árbol XML mediante la construcción funcional LINQ to XML:

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144",
                new XAttribute("Type", "Home")),
            new XElement("phone", "425-555-0145",
                new XAttribute("Type", "Work")),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
```

Tenga en cuenta que si se aplica la sangría al código para crear el árbol XML, se mostrará la estructura XML subyacente.

Para obtener más información, vea Creating XML Trees (C#) (Crear árboles XML (C#))

Trabajar directamente con elementos XML

La programación con XML suele centrarse en elementos XML y quizás en los atributos. En LINQ to XML, se puede trabajar directamente con atributos y elementos XML. Por ejemplo, puede realizar lo siguiente:

- Crear elementos XML sin usar un objeto de documento. Esto simplifica la programación cuando se tiene que trabajar con fragmentos de árboles XML.
- Cargar objetos `T:System.Xml.Linq.XElement` directamente de un archivo XML.
- Serializar objetos `T:System.Xml.Linq.XElement` a un archivo o una secuencia.

Compare esto con modelo DOM del consorcio W3C, en el que el documento XML se usa como contenedor lógico para el árbol XML. En DOM, los nodos XML, incluyendo elementos y atributos, se deben crear en el contexto de un documento XML. A continuación se muestra un fragmento del código para crear un nombre de elemento en DOM:

```
XmlDocument doc = new XmlDocument();
XmlElement name = doc.CreateElement("Name");
name.InnerText = "Patrick Hines";
doc.AppendChild(name);
```

Si desea usar un elemento en varios documentos, debe importar los nodos en los documentos. LINQ to XML evita este grado de complejidad.

Cuando se utiliza LINQ to XML, se usa la clase XDocument solamente si se desea agregar un comentario o una instrucción de procesamiento en el nivel de raíz del documento.

Control simplificado de nombres y espacios de nombres

Controlar nombres, espacios de nombres y prefijos de espacios de nombres suele ser una parte compleja de la programación XML. LINQ to XML simplifica los nombres y los espacios de nombres al eliminar el requisito de tener que tratar con prefijos de espacios de nombres. Si lo desea, puede controlar los prefijos de espacios de nombres. Pero si decide no controlar explícitamente los prefijos de espacios de nombres, durante la serialización LINQ to XML asignará prefijos de espacios de nombres si son necesarios, o serializará con espacios de nombres predeterminados si no lo son. Si se usan espacios de nombres predeterminados, no habrá prefijos de espacios de nombres en el documento resultante. Para más información, consulte Información general sobre los espacios de

nombres (LINQ to XML) (C#).

Otro problema de DOM consiste en que no permite cambiar el nombre de un nodo. Por el contrario, hay que crear un nuevo nodo y copiar en él todos los nodos secundarios, por lo que se pierde la identidad del nodo original. LINQ to XML evita este problema al permitir que se establezca la propiedad XName en un nodo.

Compatibilidad con el método estático para cargar XML

LINQ to XML permite cargar XML mediante métodos estáticos en lugar de métodos de instancia. Esto simplifica la carga y el análisis. Para más información, consulte Procedimiento para cargar un documento XML desde un archivo (C#).

Eliminación de la compatibilidad con construcciones DTD

LINQ to XML simplifica aún más la programación XML quitando la compatibilidad con entidades y referencias a entidades. La administración de entidades es compleja y su uso es muy poco común. La eliminación de la compatibilidad aumenta el rendimiento y simplifica la interfaz de programación. Cuando se rellena un árbol de LINQ to XML, se expanden todas las entidades DTD.

Compatibilidad con fragmentos

LINQ to XML no proporciona un equivalente para la clase `XmlDocumentFragment`. En cambio, es bastante común que el concepto `XmlDocumentFragment` se pueda controlar mediante el resultado de una consulta que se escribe como IEnumerable<T> de XNode o IEnumerable<T> de XElement.

Compatibilidad con XPathNavigator

LINQ to XML proporciona compatibilidad con XPathNavigator mediante los métodos de extensión del espacio de nombres System.Xml.XPath. Para obtener más información, vea System.Xml.XPath.Extensions.

Compatibilidad con espacios en blanco y sangría

LINQ to XML trata los espacios en blanco de forma más sencilla que DOM.

Un caso muy común es aquel en el que se leen datos XML con sangría, se crea un árbol XML en memoria sin ningún nodo de texto con espacios en blanco (es decir, sin preservar los espacios en blanco), se realizan ciertas operaciones sobre el XML y éste se guarda con sangría. Si se serializa el XML con formato, solo se preservan en el XML aquellos espacios en blanco más significativos. Éste es el comportamiento predeterminado en LINQ to XML.

Otro escenario muy común es aquel en el que se lee y se modifica código XML en el que se ha aplicado sangría de forma intencionada. Es posible que no desee modificar esta sangría de ninguna forma. En LINQ to XML, puede conseguirlo si preserva los espacios en blanco a la hora de cargar o analizar el XML y si deshabilita el formato cuando serialice el XML.

LINQ to XML almacena un espacio en blanco como un nodo XText, en lugar de tener un tipo de nodo Whitespace especializado, al igual que DOM.

Compatibilidad con anotaciones

Los elementos de LINQ to XML admiten un conjunto extensible de anotaciones. Esto puede resultar útil para realizar un seguimiento de información diversa de un elemento, como la información de esquema, información acerca de si un elemento está enlazado a una interfaz de usuario o cualquier otro tipo de información específica de una aplicación. Para obtener más información, vea Anotaciones en LINQ to XML.

Compatibilidad con la información de esquema

LINQ to XML proporciona compatibilidad con la validación XSD mediante métodos de extensión en el espacio de nombres System.Xml.Schema. Puede validar que un árbol XML sea compatible con un XSD. Puede rellenar el árbol XML con el conjunto de información posterior a la validación del esquema (PSVI). Para más información, consulte Procedimiento para validar con XSD (LINQ to XML) (C#) y Extensions.

Vea también

- Introducción (LINQ to XML)

LINQ to XML frente a otras tecnologías XML

18/03/2020 • 7 minutes to read • Edit Online

Este tema compara LINQ to XML con las siguientes tecnologías XML: XmlReader, XSLT, MSXML y XmlLite. Esta información puede ayudarle a decidir la tecnología que utilizará.

Para ver una comparación de LINQ to XML con Document Object Model (DOM), vea LINQ to XML frente a DOM (C#).

LINQ to XML frente a XmlReader

XmlReader es un analizador rápido, de solo avance y sin almacenamiento en caché.

LINQ to XML se implementa sobre XmlReader y ambos están estrechamente integrados. Sin embargo, puede utilizar también XmlReader por sí solo.

Por ejemplo, supongamos que está creando un servicio Web que analizará cientos de documentos XML por segundo y los documentos tienen la misma estructura, lo que significa que solo tiene que escribir una implementación de código para analizar el XML. En ese caso, probablemente deseará usar XmlReader de forma independiente.

Por el contrario, si crea un sistema que analiza muchos documentos XML más pequeños y cada uno es diferente, quizás quiera aprovechar las mejoras de productividad que LINQ to XML proporciona.

LINQ to XML frente a XSLT

Tanto LINQ to XML como XSLT proporcionan amplias capacidades de transformación de documentos XML. XSLT es un enfoque declarativo basado en reglas. Los programadores de XSLT avanzado escriben XSLT en un estilo de programación funcional que resalta un enfoque sin estado. Las transformaciones se pueden escribir con funciones puras que se implementan sin efectos secundarios. Este enfoque basado en reglas o funcional no es familiar para muchos desarrolladores y su aprendizaje puede requerir bastante tiempo.

XSLT puede ser un sistema muy productivo que produce aplicaciones de alto rendimiento. Por ejemplo, algunas grandes compañías web utilizan XSLT como forma de generar HTML a partir de XML extraído de diversos almacenes de datos. El motor XSLT administrado compila código XSLT a CLR y tiene un rendimiento incluso mejor en algunos escenarios que el motor XSLT nativo.

No obstante, XSLT no aprovecha el conocimiento de C# y Visual Basic que muchos desarrolladores tienen. Requiere que los desarrolladores escriban código en un lenguaje de programación complejo y diferente. Usar dos sistemas de desarrollo diferentes no integrados como C# (o Visual Basic) y XSLT tiene como resultado sistemas de software que son más difíciles de desarrollar y mantener.

Cuando se dominan las expresiones de consulta LINQ to XML, las transformaciones de LINQ to XML son una tecnología eficaz y fácil de usar. Básicamente, se forman documentos XML utilizando construcciones funcionales, extrayendo datos de varios orígenes, construyendo objetos XElement dinámicamente y ensamblando el conjunto en un nuevo árbol XML. La transformación puede generar un documento completamente nuevo. La construcción de transformaciones en LINQ to XML es relativamente sencilla e intuitiva y el código resultante es legible. Esto reduce los costos de desarrollo y mantenimiento.

LINQ to XML no se ha diseñado para sustituir a XSLT. XSLT sigue siendo la herramienta preferida de transformaciones XML complicadas y basadas en documentos, especialmente si la estructura del documento no está bien definida.

XSLT tiene la ventaja de que es un estándar de World Wide Web Consortium (W3C). Si tiene el requisito de usar solamente tecnologías que son estándares, XSLT puede ser más adecuado.

XSLT es XML y, por lo tanto, se puede manipular mediante programación.

LINQ to XML frente a MSXML

MSXML es la tecnología basada en COM utilizada para procesar XML y que se incluye con Microsoft Windows. MSXML ofrece una implementación nativa de DOM, que es compatible con XPath y XSLT. También contiene el analizador basado en eventos y sin almacenamiento en caché SAX2.

MSXML tiene un buen rendimiento, es seguro de forma predeterminada en la mayoría de casos y se puede tener acceso a él en Internet Explorer para realizar procesamiento XML en el cliente en aplicaciones de estilo AJAX. MSXML se puede usar en cualquier lenguaje de programación que admita COM, incluidos C++, JavaScript y Visual Basic 6.0.

No se recomienda el uso de MSXML en código administrado basado en Common Language Runtime (CLR).

LINQ to XML frente a XmlLite

XmlLite es un analizador de extracción sin almacenamiento en caché, de solo avance. Los desarrolladores utilizan principalmente XmlLite con C++. No se recomienda a los desarrolladores usar XmlLite con código administrado.

La principal ventaja de XmlLite es que es un analizador XML rápido y ligero que es seguro en la mayoría de casos. Su área de exposición a amenazas es muy pequeña. Si debe analizar documentos que no son de confianza y desea protegerse de ataques tales como denegación de servicio o exposición de datos, XmlLite puede ser una buena opción.

XmlLite no está integrado con Language-Integrated Query (LINQ). No proporciona al programador mejoras de productividad que sean la fuerza motivadora detrás de LINQ.

Vea también

- Introducción (LINQ to XML)

Programación funcional frente a programación basada en procedimientos (LINQ to XML) (C#)

18/03/2020 • 3 minutes to read • Edit Online

Existen varios tipos de aplicaciones XML:

- Algunas aplicaciones toman los documentos XML de origen y crean nuevos documentos XML que tienen una forma diferente que los documentos de origen.
- Algunas aplicaciones toman documentos XML de origen y crean documentos de resultado de una forma totalmente diferente, como archivos HTML o de texto CSV.
- Algunas aplicaciones toman documentos XML de origen e insertan registros en una base de datos.
- Algunas aplicaciones toman datos de otro origen de datos, como una base de datos y crean documentos XML a partir de él.

Estos no son todos los tipos de aplicaciones XML, pero son un conjunto representativo de los tipos de funcionalidad que un programador XML debe implementar.

Con todos esos tipos de aplicaciones un desarrollador puede tomar dos enfoques opuestos:

- Construcción funcional usando un enfoque declarativo.
- Modificación del árbol XML en memoria usando código de procedimiento.

LINQ to XML admite ambos enfoques.

Cuando se utiliza el enfoque funcional, se escriben transformaciones que toman los documentos de origen y generan documentos de resultados completamente nuevos con la forma que se desee.

Cuando se modifica un árbol XML, se escribe código que atraviese los nodos de un árbol XML y navegue por ellos en memoria para insertar, eliminar y modificar nodos según sea necesario.

Puede usar LINQ to XML con cualquier enfoque. Se usan las mismas clases y, en algunos casos, los mismos métodos. No obstante, la estructura y los objetivos de los dos enfoques son muy diferentes. Por ejemplo, en situaciones diferentes, uno u otro enfoque tendrá a menudo un rendimiento mejor y usará más o menos memoria. Asimismo, uno u otro enfoque será más fácil de escribir y proporcionará código más fácil de mantener.

Para obtener una comparación de los dos enfoques, vea Diferencias entre la modificación del árbol XML en memoria y la construcción funcional (LINQ to XML) (C#).

Para obtener un tutorial sobre la escritura de transformaciones funcionales, vea Pure Functional Transformations of XML (C#) (Transformaciones funcionales puras de XML (C#)).

Vea también

- Información general sobre la programación de LINQ to XML (C#)

Información general de las clases LINQ to XML (C#)

18/03/2020 • 6 minutes to read • Edit Online

En este tema se proporciona una lista de las clases de LINQ to XML en el espacio de nombres System.Xml.Linq y proporciona una breve descripción de cada una.

Clases de LINQ to XML

Clase XAttribute

XAttribute representa un atributo XML. Para obtener información detallada y ejemplos, vea Información general acerca de la clase XAttribute.

Clase XCData

XCData representa un nodo de texto CDATA.

Clase XComment

XComment representa un comentario XML.

Clase XContainer

XContainer es una clase base abstracta para todos los nodos que pueden tener nodos secundarios. Las siguientes clases se derivan de la clase XContainer:

- XElement
- XDocument

Clase XDeclaration

XDeclaration representa una declaración XML. Una declaración XML se utiliza para declarar la versión de XML y la codificación de un documento. Asimismo, una declaración XML especifica si el documento XML es independiente. Si un documento es independiente, no hay declaraciones de marcado externas, ya sea en un DTD externo o en una entidad de parámetro externa a la que se hace referencia desde el subconjunto interno.

Clase XDocument

XDocument representa un documento XML. Para obtener información detallada y ejemplos, vea Información general acerca de la clase XDocument.

Clase XDocumentType

XDocumentType representa una definición de tipo del documento (DTD) XML.

Clase XElement

XElement representa un elemento XML. Para obtener información detallada y ejemplos, vea Información general acerca de la clase XElement.

Clase XName

XName representa nombres de elementos (XElement) y atributos (XAttribute). Para obtener información detallada y ejemplos, vea Información general acerca de la clase XDocument.

LINQ to XML se ha diseñado para hacer que los nombres XML sean tan sencillos como sea posible. Debido a su complejidad, los nombres XML a menudo se consideran un tema avanzado en XML. Puede argumentarse que la complejidad no proviene de los espacios de nombres, que los desarrolladores usan regularmente en la programación, sino de los prefijos de los espacios de nombres. Los prefijos de los espacios de nombres pueden ser útiles para reducir las pulsaciones de teclas necesarias cuando se especifica código XML o hacer que el código XML

sea más fácil de leer. No obstante, a menudo los prefijos son un acceso directo al espacio de nombres XML completo y no son necesarios en la mayoría de los casos. LINQ to XML simplifica los nombres XML resolviendo todos los prefijos a su espacio de nombres XML correspondiente. Los prefijos están disponibles, si son necesarios, a través del método GetPrefixOfNamespace.

Si es necesario es posible controlar los prefijos de espacios de nombres. En algunas circunstancias, si trabaja con otros sistemas XML, como XSLT o XAML, debe controlar los prefijos de espacio de nombres. Por ejemplo, si tiene una expresión XPath que utiliza los prefijos del espacio de nombres que está incrustado en una hoja de estilos XSLT, deberá asegurarse de que el documento XML esté serializado con prefijos de espacio de nombres que coinciden con los que se utilizan en la expresión XPath.

Clase XNamespace

XNamespace representa un espacio de nombres para un XElement o XAttribute. Los espacios de nombres son un componente de un XName.

Clase XNode

XNode es una clase abstracta que representa los nodos de un árbol XML. Las siguientes clases se derivan de la clase XNode:

- XText
- XContainer
- XComment
- XProcessingInstruction
- XDocumentType

Clase XNodeDocumentOrderComparer

XNodeDocumentOrderComparer proporciona la funcionalidad para comparar el orden de documentos de los nodos.

Clase XNodeEqualityComparer

XNodeEqualityComparer proporciona la funcionalidad para comparar la igualdad del valor de los nodos.

Clase XObject

XObject es una clase base abstracta de XNode y XAttribute. Proporciona funcionalidad de evento y anotación.

Clase XObjectChange

XObjectChange especifica el tipo de evento cuando se produce para un XObject.

Clase XObjectChangeEventArgs

XObjectChangeEventArgs proporciona datos de los eventos Changing y Changed.

Clase XProcessingInstruction

XProcessingInstruction representa una instrucción de procesamiento de XML. Una instrucción de procesamiento comunica información a una aplicación que procesa el XML.

Clase XText

XText representa un nodo de texto. En la mayoría de casos no tiene que usar esta clase. Esta clase se utiliza principalmente para el contenido mixto.

Vea también

- Información general sobre la programación de LINQ to XML (C#)

Información general de la clase XElement (C#)

18/03/2020 • 4 minutes to read • Edit Online

La clase XElement es una de las clases fundamentales de LINQ to XML. Representa a un elemento XML. Puede utilizar esta clase para crear elementos; cambiar el contenido del elemento; agregar, modificar o eliminar elementos secundarios; agregar atributos a un elemento; o serializar el contenido de un elemento en forma de texto. También puede operar con otras clases de System.Xml, como son XmlReader, XmlWriter y XslCompiledTransform.

Este tema describe la funcionalidad que ofrece la clase XElement.

Construir árboles XML

Es posible construir árboles XML de diferentes formas, entre las que se incluyen las siguientes:

- Puede construir un árbol XML mediante código. Para obtener más información, consulte Crear árboles XML (C#).
- Puede analizar XML a partir de diferentes orígenes, incluyendo un TextReader, archivos de texto o direcciones web (URL). Para obtener más información, consulte Analizar XML (C#).
- También puede utilizar un XmlReader para rellenar el árbol. Para obtener más información, consulta ReadFrom.
- Si dispone de un módulo que pueda escribir contenidos en un XmlWriter, puede utilizar el método CreateWriter para crear un sistema de escritura, para pasar éste al módulo y para utilizar después el contenido que se haya escrito en XmlWriter para rellenar el árbol XML.

No obstante, la forma más habitual de crear un árbol XML es la siguiente:

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144"),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
```

Otra técnica que se usa con frecuencia para crear un árbol XML implica el uso de los resultados de una consulta LINK para rellenar el árbol XML, tal y como se muestra en el ejemplo siguiente:

```
XElement srcTree = new XElement("Root",
    new XElement("Element", 1),
    new XElement("Element", 2),
    new XElement("Element", 3),
    new XElement("Element", 4),
    new XElement("Element", 5)
);
XElement xmlTree = new XElement("Root",
    new XElement("Child", 1),
    new XElement("Child", 2),
    from el in srcTree.Elements()
    where (int)el > 2
    select el
);
Console.WriteLine(xmlTree);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <Child>1</Child>
  <Child>2</Child>
  <Element>3</Element>
  <Element>4</Element>
  <Element>5</Element>
</Root>
```

Serializar árboles XML

Puede serializar un árbol XML en un File, un TextWriter o en un XmlWriter.

Para obtener más información, consulte Serializar árboles XML (C#).

Recuperar datos XML mediante los métodos de los ejes

Puede utilizar los métodos de los ejes para recuperar atributos, elementos secundarios, elementos descendientes y elementos antecesores. Las consultas LINK operan sobre métodos de eje y proporcionan varias formas flexibles y eficaces de recorrer y procesar árboles XML.

Para obtener más información, consulte Ejes de LINQ to XML (C#).

Consultar árboles XML

Puede escribir consultas LINK que extraigan datos de un árbol XML.

Para obtener más información, consulte Consultar árboles XML (C#).

Modificar árboles XML

Puede modificar un elemento de diferentes maneras, incluyendo el cambiar su contenido o sus atributos. También puede eliminar un elemento dependiente de un elemento primario.

Para obtener más información, consulte Modificar árboles XML (LINQ to XML) (C#).

Vea también

- Información general sobre la programación de LINQ to XML (C#)

Información general de la clase XAttribute (C#)

18/03/2020 • 3 minutes to read • Edit Online

Los atributos son pares de nombre y valor asociados a un elemento. La clase XAttribute representa los atributos XML.

Información general

Trabajar con atributos en LINQ to XML es similar a trabajar con elementos. Sus constructores son similares. Los métodos que usa para recuperar colecciones de ellos también son similares. Una expresión de consulta LINQ de una colección de atributos se parece mucho a una expresión de consulta LINQ de una colección de elementos.

Se conserva el orden en el que se agregaron los atributos a un elemento. Es decir, cuando procese una iteración en los atributos, los verá en el mismo orden en el que se agregaron.

El constructor XAttribute

El siguiente constructor de la clase XAttribute es el que usará normalmente:

CONSTRUCTOR	DESCRIPTION
`XAttribute(XName name, object content)`	Crea un objeto XAttribute. El argumento `name` especifica el nombre del atributo; `content` especifica el contenido del atributo.

Crear un elemento con un atributo

El siguiente código muestra la tarea habitual de crear un elemento que contiene un atributo:

```
XElement phone = new XElement("Phone",
    new XAttribute("Type", "Home"),
    "555-555-5555");
Console.WriteLine(phone);
```

Este ejemplo produce el siguiente resultado:

```
<Phone Type="Home">555-555-5555</Phone>
```

Construcción funcional de los atributos

Puede construir objetos XAttribute de modo similar a la construcción de los objetos XElement, tal como se indica a continuación:

```
XElement c = new XElement("Customers",
    new XElement("Customer",
        new XElement("Name", "John Doe"),
        new XElement("PhoneNumbers",
            new XElement("Phone",
                new XAttribute("type", "home"),
                "555-555-5555"),
            new XElement("Phone",
                new XAttribute("type", "work"),
                "666-666-6666")
        )
    )
);
Console.WriteLine(c);
```

Este ejemplo produce el siguiente resultado:

```
<Customers>
  <Customer>
    <Name>John Doe</Name>
    <PhoneNumbers>
      <Phone type="home">555-555-5555</Phone>
      <Phone type="work">666-666-6666</Phone>
    </PhoneNumbers>
  </Customer>
</Customers>
```

Los atributos no son nodos

Existen algunas diferencias entre los atributos y los elementos. Los objetos XAttribute no son nodos en el árbol XML. Son pares de nombre y valor asociados a un elemento XML. A diferencia de Document Object Model (DOM), esto refleja de manera más precisa la estructura del código XML. Aunque los objetos XAttribute no son realmente nodos en el árbol XML, trabajar con objetos XAttribute es muy similar a trabajar con objetos XElement.

Esta distinción solo resulta de importancia para los desarrolladores que escriban código que trabaje con árboles XML en el nivel de nodo. A muchos desarrolladores no les afecta esta distinción.

Vea también

- Información general sobre la programación de LINQ to XML (C#)

Información general de la clase XDocument (C#)

18/03/2020 • 4 minutes to read • Edit Online

En este tema se presenta la clase XDocument.

Información general acerca de la clase XDocument

La clase XDocument contiene la información necesaria para un documento XML válido. Incluye una declaración XML, instrucciones de procesamiento y comentarios.

Tenga en cuenta que solo debe crear objetos XDocument si necesita la funcionalidad específica que proporciona la clase XDocument. En muchas circunstancias, puede trabajar directamente con XElement. Trabajar directamente con XElement constituye un modelo de programación más simple.

XDocument se deriva de XContainer. Por lo tanto, puede contener nodos secundarios. Sin embargo, los objetos XDocument solo pueden tener un nodo XElement secundario. Esto refleja el estándar XML: solo puede haber un elemento raíz en un documento XML.

Componentes de XDocument

Un objeto XDocument puede contener los siguientes elementos:

- Un objeto XDeclaration. XDeclaration permite especificar las partes pertinentes de una declaración XML: la versión XML, la codificación del documento y si el documento XML es independiente.
- Un objeto XElement. Es el nodo raíz del documento XML.
- Cualquier número de objetos XProcessingInstruction. Una instrucción de procesamiento comunica información a una aplicación que procesa el XML.
- Cualquier número de objetos XComment. Los comentarios serán del mismo nivel que el elemento raíz. El objeto XComment no puede ser el primer argumento de la lista, ya que no es válido que el documento XML empiece con un comentario.
- Un elemento XDocumentType para el DTD.

Al serializar un objeto XDocument, aunque `XDocument.Declaration` sea `null`, el resultado generará una declaración XML, siempre y cuando el redactor haya establecido `Writer.Settings.OmitXmlDeclaration` en `false` (valor predeterminado).

De manera predeterminada, LINQ to XML establece la versión en "1.0" y la codificación en "utf-8".

Usar XElement sin XDocument

Tal como se indicó anteriormente, la clase XElement es la clase principal de la interfaz de programación LINQ to XML. En muchos casos, la aplicación no requerirá la creación de un documento. Mediante la clase XElement, puede crear un árbol XML, agregarle otros árboles XML, modificar el árbol XML o guardarlo.

Usar XDocument

Para construir un objeto XDocument, use la construcción funcional, al igual que cuando se construyen objetos XElement.

El código siguiente crea un objeto XDocument y sus objetos contenidos asociados.

```
XDocument d = new XDocument(
    new XComment("This is a comment."),
    new XProcessingInstruction("xml-stylesheet",
        "href='mystyle.css' title='Compact' type='text/css'"),
    new XElement("Pubs",
        new XElement("Book",
            new XElement("Title", "Artifacts of Roman Civilization"),
            new XElement("Author", "Moreno, Jordao")
        ),
        new XElement("Book",
            new XElement("Title", "Midieval Tools and Implements"),
            new XElement("Author", "Gazit, Inbar")
        )
    ),
    new XComment("This is another comment.")
);
d.Declaration = new XDeclaration("1.0", "utf-8", "true");
Console.WriteLine(d);

d.Save("test.xml");
```

Al examinar el archivo test.xml, obtendrá el siguiente resultado:

```
<?xml version="1.0" encoding="utf-8"?>
<!--This is a comment.-->
<?xml-stylesheet href='mystyle.css' title='Compact' type='text/css'?>
<Pubs>
  <Book>
    <Title>Artifacts of Roman Civilization</Title>
    <Author>Moreno, Jordao</Author>
  </Book>
  <Book>
    <Title>Midieval Tools and Implements</Title>
    <Author>Gazit, Inbar</Author>
  </Book>
</Pubs>
<!--This is another comment.-->
```

Vea también

- Información general sobre la programación de LINQ to XML (C#)

Procedimiento para compilar ejemplos de LINQ to XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

Los diferentes fragmentos de código y ejemplos de esta documentación utilizan clases y tipos de diferentes espacios de nombres. Cuando se compila código C#, se deben proporcionar las directivas `using` adecuadas.

Ejemplo

El siguiente código contiene las directivas `using` que el ejemplo de C# requiere para crearse y ejecutarse. No todas las directivas `using` son necesarias para cada ejemplo.

```
using System;
using System.Diagnostics;
using System.Collections;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Text;
using System.Linq;
using System.Xml;
using System.Xml.Linq;
using System.Xml.Schema;
using System.Xml.XPath;
using System.Xml.Xsl;
using System.IO;
using System.Threading;
using System.Reflection;
using System.IO.Packaging;
```

Vea también

- Información general sobre la programación de LINQ to XML (C#)

Construcción funcional (LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

LINQ to XML proporciona una manera eficaz de crear elementos XML denominada *construcción funcional*. La construcción funcional es la capacidad de crear un árbol XML en una sola instrucción.

Hay varias características fundamentales de la interfaz de programación de LINQ to XML que permiten la construcción funcional:

- El constructor XElement toma varios tipos de argumentos para el contenido. Por ejemplo, puede pasar otro objeto XElement, que se convierte en un elemento secundario. Puede pasar un objeto XAttribute, que se convierte en un atributo del elemento. O bien, puede pasar cualquier otro tipo de objeto, que se convierte en una cadena y en el contenido de texto del elemento.
- El constructor XElement toma una matriz de `params` del tipo Object, de forma que puede pasar cualquier número de objetos al constructor. Esto permite crear un elemento que tiene un contenido complejo.
- Si un objeto implementa IEnumerable<T>, se enumera la colección del objeto y se agregan todos los elementos de la colección. Si la colección contiene objetos XElement o XAttribute, cada elemento de la colección se agrega por separado. Esto es importante porque le permite pasar los resultados de una consulta LINQ al constructor.

Estas características permiten escribir código para crear un árbol XML. A continuación se muestra un ejemplo:

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144"),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
```

Estas características también permiten escribir código que usa los resultados de las consultas LINQ cuando crea un árbol XML, de la siguiente manera:

```
XElement srcTree = new XElement("Root",
    new XElement("Element", 1),
    new XElement("Element", 2),
    new XElement("Element", 3),
    new XElement("Element", 4),
    new XElement("Element", 5)
);
XElement xmlTree = new XElement("Root",
    new XElement("Child", 1),
    new XElement("Child", 2),
    from el in srcTree.Elements()
    where (int)el > 2
    select el
);
Console.WriteLine(xmlTree);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <Child>1</Child>
  <Child>2</Child>
  <Element>3</Element>
  <Element>4</Element>
  <Element>5</Element>
</Root>
```

Crear árboles XML en C# (LINQ to XML)

18/03/2020 • 8 minutes to read • Edit Online

En esta sección encontrará información acerca de cómo crear árboles XML en C#.

Para obtener información sobre cómo usar los resultados de las consultas LINQ como contenido de un XElement, vea Construcción funcional (LINQ to XML) (C#).

Elementos de construcción

Las firmas de los constructores XElement y XAttribute le permiten pasar los contenidos del elemento o atributo como argumentos del constructor. Dado que uno de los constructores recibe un número variable de argumentos, podrá pasar tantos elementos secundarios como desee. Por supuesto, cada uno de esos elementos secundarios podrá contener sus propios elementos secundarios. Para cualquier elemento, podrá agregar cuantos atributos desee.

Cuando se agregan objetos XNode (incluyendo el objeto XElement) o XAttribute, si el contenido nuevo no tiene un elemento primario, los objetos simplemente se adjuntan al árbol XML. Si el contenido nuevo ya tiene un elemento primario y forma parte de otro árbol XML, el nuevo contenido se clonará y ese clon se asociará al árbol XML. El último ejemplo de este tema muestra este comportamiento.

Para crear un `contacts` XElement, podría utilizar el siguiente código:

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144"),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
```

Si se aplica correctamente la sangría, el código que construye los objetos XElement se asemeja mucho a la estructura del contenido XML subyacente.

Constructores de XElement

La clase XElement utiliza los siguientes constructores para llevar a cabo una construcción funcional. Observe que existen algunos constructores más para XElement, pero, dado que no se utilizan para el proceso de construcción funcional, no se detallarán aquí.

CONSTRUCTOR	DESCRIPTION
`XElement(XName name, object content)`	Crea una interfaz XElement. El parámetro `name` especifica el nombre del elemento; `content` especifica el contenido del elemento.

CONSTRUCTOR	DESCRIPTION
`XElement(XName name)`	Crea un XElement cuyo XName se inicializa con el nombre especificado.
`XElement(XName name, params object[] content)`	Crea un XElement cuyo XName se inicializa con el nombre especificado. Los atributos y/o elementos secundarios se crean a partir de los contenidos de la lista de parámetros.

El parámetro `content` es extremadamente flexible. Admite cualquier tipo de objeto que sea un elemento secundario válido de un XElement. Se aplicarán las siguientes reglas a los diferentes tipos de objetos pasados en este parámetros:

- Una cadena se agrega como contenido de tipo texto.
- Un XElement se agrega como un elemento secundario.
- Un XAttribute se agrega como un atributo.
- Un XProcessingInstruction, XComment o un XText se agregan como un contenido secundario.
- Un IEnumerable es un tipo enumerado y se aplicarán estas reglas recursivamente a los resultados.
- Para el resto de tipos, se llamará a su método `ToString` y se agregará el resultado como contenido de tipo texto.

Creación de un XElement con contenidos

Puede crear un XElement que contenga un contenido simple con una única llamada a método. Para ello, especifique el contenido como segundo parámetro, tal y como sigue:

```
XElement n = new XElement("Customer", "Adventure Works");
Console.WriteLine(n);
```

Este ejemplo produce el siguiente resultado:

```
<Customer>Adventure Works</Customer>
```

Puede pasar cualquier tipo de objeto como contenido. Por ejemplo, el siguiente código crea un elemento que contiene un número de punto flotante como contenido:

```
XElement n = new XElement("Cost", 324.50);
Console.WriteLine(n);
```

Este ejemplo produce el siguiente resultado:

```
<Cost>324.5</Cost>
```

Al número de punto flotante se le aplica la conversión boxing y se pasa al constructor. El número se convierte entonces en una cadena y se utiliza como contenido del elemento.

Creación de un XElement con un elemento secundario

Si pasa una instancia de la clase XElement para el argumento del contenido, el constructor creará un elemento que tiene un elemento secundario:

```
XElement shippingUnit = new XElement("ShippingUnit",
    new XElement("Cost", 324.50)
);
Console.WriteLine(shippingUnit);
```

Este ejemplo produce el siguiente resultado:

```
<ShippingUnit>
  <Cost>324.5</Cost>
</ShippingUnit>
```

Creación de un XElement con varios elementos secundarios

Para el contenido, puede pasar tantos objetos XElement como desee. Cada uno de los objetos XElement se incluye como elemento secundario.

```
XElement address = new XElement("Address",
    new XElement("Street1", "123 Main St"),
    new XElement("City", "Mercer Island"),
    new XElement("State", "WA"),
    new XElement("Postal", "68042")
);
Console.WriteLine(address);
```

Este ejemplo produce el siguiente resultado:

```
<Address>
  <Street1>123 Main St</Street1>
  <City>Mercer Island</City>
  <State>WA</State>
  <Postal>68042</Postal>
</Address>
```

Si se extiende el ejemplo anterior, es posible crear un árbol XML completo, tal y como sigue:

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144"),
            new XElement("Address",
                new XElement("Street1", "123 Main St"),
                new XElement("City", "Mercer Island"),
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );
Console.WriteLine(contacts);
```

Este ejemplo produce el siguiente resultado:

```
<Contacts>
  <Contact>
    <Name>Patrick Hines</Name>
    <Phone>206-555-0144</Phone>
    <Address>
      <Street1>123 Main St</Street1>
      <City>Mercer Island</City>
      <State>WA</State>
      <Postal>68042</Postal>
    </Address>
  </Contact>
</Contacts>
```

Creación de un XElement con un XAttribute

Si pasa una instancia de la clase XAttribute para el argumento del contenido, el constructor creará un elemento con un atributo:

```
XElement phone = new XElement("Phone",
    new XAttribute("Type", "Home"),
    "555-555-5555");
Console.WriteLine(phone);
```

Este ejemplo produce el siguiente resultado:

```
<Phone Type="Home">555-555-5555</Phone>
```

Creación de un elemento vacío

Para crear un XElement vacío, no pase ningún contenido al constructor. El siguiente ejemplo crea un elemento vacío:

```
XElement n = new XElement("Customer");
Console.WriteLine(n);
```

Este ejemplo produce el siguiente resultado:

```
<Customer />
```

Diferencias entre asociar y clonar

Como mencionamos anteriormente, cuando se agregan objetos XNode (incluyendo el objeto XElement) o XAttribute, si el contenido nuevo no tiene un elemento primario, los objetos simplemente se adjuntan al árbol XML. Si el contenido nuevo ya tiene un elemento primario y forma parte de otro árbol XML, el nuevo contenido se clonará y ese clon se asociará al árbol XML.

En el siguiente ejemplo se demuestra qué ocurre si agrega un elemento que tiene elemento primario a un árbol y qué ocurre si agrega un elemento que no tenga elemento primario a un árbol.

```
// Create a tree with a child element.
XElement xmlTree1 = new XElement("Root",
    new XElement("Child1", 1)
);

// Create an element that is not parented.
XElement child2 = new XElement("Child2", 2);

// Create a tree and add Child1 and Child2 to it.
XElement xmlTree2 = new XElement("Root",
    xmlTree1.Element("Child1"),
    child2
);

// Compare Child1 identity.
Console.WriteLine("Child1 was {0}",
    xmlTree1.Element("Child1") == xmlTree2.Element("Child1") ?
    "attached" : "cloned");

// Compare Child2 identity.
Console.WriteLine("Child2 was {0}",
    child2 == xmlTree2.Element("Child2") ?
    "attached" : "cloned");

// The example displays the following output:
//    Child1 was cloned
//    Child2 was attached
```

Vea también

- Crear árboles XML (C#)

Procedimiento para analizar una cadena (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo analizar una cadena para crear un árbol XML en C#.

Ejemplo

El siguiente código de C# muestra cómo analizar una cadena XML:

```
XElement contacts = XElement.Parse(
    @"<Contacts>
        <Contact>
            <Name>Patrick Hines</Name>
            <Phone Type=""home"">206-555-0144</Phone>
            <Phone Type=""work"">425-555-0145</Phone>
            <Address>
            <Street1>123 Main St</Street1>
            <City>Mercer Island</City>
            <State>WA</State>
            <Postal>68042</Postal>
            </Address>
            <NetWorth>10</NetWorth>
        </Contact>
        <Contact>
            <Name>Gretchen Rivas</Name>
            <Phone Type=""mobile"">206-555-0163</Phone>
            <Address>
            <Street1>123 Main St</Street1>
            <City>Mercer Island</City>
            <State>WA</State>
            <Postal>68042</Postal>
            </Address>
            <NetWorth>11</NetWorth>
        </Contact>
    </Contacts>");
Console.WriteLine(contacts);
```

El nodo `Contacts` raíz tiene dos nodos `Contact`. Para acceder a algunos datos específicos en su XML analizado, use el método XElement.Elements(), que en este caso devuelve los elementos secundarios del nodo raíz `Contacts`. En el siguiente ejemplo se imprime el primer nodo `Contact` en la consola:

```
List<XElement> contactNodes = contacts.Elements("Contact").ToList();
Console.WriteLine(contactNodes[0]);
```

Vea también

- Procedimiento para buscar un elemento con un atributo específico (C#)

Procedimiento para cargar un documento XML desde un archivo (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo cargar XML de una dirección URI usando el método XElement.Load.

Ejemplo

El siguiente ejemplo muestra cómo cargar un documento XML desde un archivo. El siguiente ejemplo carga books.xml y produce el árbol XML en la consola.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: libros (LINQ to XML).

```
XElement booksFromFile = XElement.Load(@"books.xml");
Console.WriteLine(booksFromFile);
```

Este código genera el siguiente resultado:

```
<Catalog>
  <Book id="bk101">
    <Author>Garghentini, Davide</Author>
    <Title>XML Developer's Guide</Title>
    <Genre>Computer</Genre>
    <Price>44.95</Price>
    <PublishDate>2000-10-01</PublishDate>
    <Description>An in-depth look at creating applications
      with XML.</Description>
  </Book>
  <Book id="bk102">
    <Author>Garcia, Debra</Author>
    <Title>Midnight Rain</Title>
    <Genre>Fantasy</Genre>
    <Price>5.95</Price>
    <PublishDate>2000-12-16</PublishDate>
    <Description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</Description>
  </Book>
</Catalog>
```

Mantener un espacio en blanco al cargar o analizar XML

18/03/2020 • 3 minutes to read • Edit Online

En este tema, se describe cómo controlar el comportamiento de los espacios en blanco de LINQ to XML.

Un caso muy común es aquel en el que se leen datos XML con sangría, se crea un árbol XML en memoria sin ningún nodo de texto con espacios en blanco (es decir, sin preservar los espacios en blanco), se realizan ciertas operaciones sobre el XML y éste se guarda con sangría. Si se serializa el XML con formato, solo se preservan en el XML aquellos espacios en blanco más significativos. Éste es el comportamiento predeterminado en LINQ to XML.

Otro escenario muy común es aquel en el que se lee y se modifica código XML en el que se ha aplicado sangría de forma intencionada. Es posible que no desee modificar esta sangría de ninguna forma. Para hacerlo en LINQ to XML, puede preservar los espacios en blanco a la hora de cargar o analizar el XML y si deshabilita el formato cuando serialice el XML.

En este tema, se describe el comportamiento de espacios en blanco de métodos que rellenan los árboles XML. Para obtener información sobre cómo controlar los espacios en blanco al serializar árboles XML, consulte Mantener un espacio en blanco al serializar.

Comportamiento de métodos que rellenan árboles XML

Los siguientes métodos de las clases XElement y XDocument rellenan un árbol XML. Puede rellenar un árbol XML desde un archivo, un TextReader, un XmlReader o una cadena:

- XElement.Load
- XElement.Parse
- XDocument.Load
- XDocument.Parse

Si el método no toma LoadOptions como argumento, el método no conservará espacios en blanco no significativos.

En la mayoría de casos, si el método toma LoadOptions como argumento, opcionalmente se pueden conservar los espacios en blanco no significativos como nodos de texto en el árbol XML. No obstante, si el método carga XML desde XmlReader, XmlReader determina si los espacios en blanco se conservan. Establecer PreserveWhitespace no tendrá ningún efecto.

Con estos métodos, si se conservan los espacios en blanco, se inserta un espacio en blanco insignificante en el árbol XML como nodos XText. Si no se conservan los espacios en blanco, entonces no se insertan los nodos de texto.

Puede crear un árbol XML mediante XmlWriter. Los nodos escritos en XmlWriter se rellenan en el árbol. No obstante, al crear un árbol XML usando este método se conservan todos los nodos, independientemente de si el nodo es o no un espacio en blanco, o si el espacio en blanco es o no significativo.

Procedimiento para detectar errores de análisis (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se describe cómo detectar XML no válido o mal formado.

LINQ to XML se implementa con XmlReader. Si se pasa XML no válido o mal formado a LINQ to XML, la clase XmlReader subyacente iniciará una excepción. Los diferentes métodos que analizan XML, como XElement.Parse no detectan la excepción; la excepción se propaga de forma que la aplicación pueda detectarla.

Ejemplo

El siguiente código intenta analizar XML no válido:

```
try {
    XElement contacts = XElement.Parse(
        @"<Contacts>
            <Contact>
                <Name>Jim Wilson</Name>
            </Contact>
          </Contcts>");

    Console.WriteLine(contacts);
}
catch (System.Xml.XmlException e)
{
    Console.WriteLine(e.Message);
}
```

Cuando ejecuta este código, devuelve la siguiente excepción:

```
The 'Contacts' start tag on line 1 does not match the end tag of 'Contcts'. Line 5, position 13.
```

Para obtener información acerca de las excepciones que puede esperar que devuelva los métodos XElement.Parse, XDocument.Parse, XElement.Load y XDocument.Load, vea la documentación de XmlReader.

Procedimiento para crear un árbol a partir de un objeto XmlReader (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo crear un árbol XML directamente de XmlReader. Para crear un XElement de XmlReader, debe colocar el XmlReader en un nodo de elemento. XmlReader omitirá los comentarios y las instrucciones de procesamiento, pero si XmlReader se coloca en un nodo de texto, se producirá un error. Para evitar tales errores, coloque siempre XmlReader en un elemento ante de crear un árbol XML de XmlReader.

Ejemplo

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: libros (LINQ to XML).

El siguiente código crea un objeto `T:System.Xml.XmlReader` y lee nodos hasta que encuentra el primer nodo de elemento. A continuación, carga el objeto XElement.

```
XmlReader r = XmlReader.Create("books.xml");
while (r.NodeType != XmlNodeType.Element)
    r.Read();
XElement e = XElement.Load(r);
Console.WriteLine(e);
```

Este ejemplo produce el siguiente resultado:

```
<Catalog>
   <Book id="bk101">
      <Author>Garghentini, Davide</Author>
      <Title>XML Developer's Guide</Title>
      <Genre>Computer</Genre>
      <Price>44.95</Price>
      <PublishDate>2000-10-01</PublishDate>
      <Description>An in-depth look at creating applications
      with XML.</Description>
   </Book>
   <Book id="bk102">
      <Author>Garcia, Debra</Author>
      <Title>Midnight Rain</Title>
      <Genre>Fantasy</Genre>
      <Price>5.95</Price>
      <PublishDate>2000-12-16</PublishDate>
      <Description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</Description>
   </Book>
</Catalog>
```

Vea también

- Parsing XML (C#) (Analizar XML [C#])

Streaming de fragmentos XML desde un objeto XmlReader (C#)

18/03/2020 • 3 minutes to read • Edit Online

Cuando deba procesar archivos XML grandes quizás no sea factible cargar la totalidad del árbol XML en memoria. En este tema se muestra cómo transmitir por secuencias fragmentos usando XmlReader.

Una de las formas más efectivas de usar XmlReader para leer objetos XElement es escribir un método de eje personalizado propio. Un método de eje suele devolver una recopilación como IEnumerable<T> de XElement, tal y como se muestra en el ejemplo de este tema. En el método de eje personalizado, tras crear el fragmento XML llamando al método ReadFrom, devuelva la recopilación usando `yield return`. Esto proporciona semántica de ejecución aplazada al método de eje personalizado.

Cuando crea un árbol XML de un objeto XmlReader, XmlReader debe estar posicionado en un elemento. El método ReadFrom no vuelve hasta que ha leído la etiqueta de cierre del elemento.

Si desea crear un árbol parcial, puede crear una instancia de un XmlReader, colocar el lector en el nodo que desea convertir a un árbol XElement y después crear el objeto XElement.

El tema Procedimiento para hacer streaming de fragmentos XML con acceso a la información del encabezado (C#) contiene información y un ejemplo sobre cómo hacer streaming de un documento más complejo.

El tema Procedimiento para realizar una transformación de streaming de documentos XML grandes (C#) contiene un ejemplo del uso de LINQ to XML para transformar documentos XML extremadamente grandes manteniendo una superficie de memoria pequeña.

Ejemplo

Este ejemplo crea un método de eje personalizado. Puede consultarlo usando una consulta de LINQ. El método de eje personalizado, `StreamRootChildDoc`, es un método que está específicamente diseñado para leer un documento que tiene un elemento `Child` que se repite.

```
static IEnumerable<XElement> StreamRootChildDoc(StringReader stringReader)
{
    using (XmlReader reader = XmlReader.Create(stringReader))
    {
        reader.MoveToContent();
        // Parse the file and display each of the nodes.
        while (reader.Read())
        {
            switch (reader.NodeType)
            {
                case XmlNodeType.Element:
                    if (reader.Name == "Child") {
                        XElement el = XElement.ReadFrom(reader) as XElement;
                        if (el != null)
                            yield return el;
                    }
                    break;
            }
        }
    }
}

static void Main(string[] args)
{
    string markup = @"<Root>
      <Child Key=""01"">
        <GrandChild>aaa</GrandChild>
      </Child>
      <Child Key=""02"">
        <GrandChild>bbb</GrandChild>
      </Child>
      <Child Key=""03"">
        <GrandChild>ccc</GrandChild>
      </Child>
    </Root>";

    IEnumerable<string> grandChildData =
        from el in StreamRootChildDoc(new StringReader(markup))
        where (int)el.Attribute("Key") > 1
        select (string)el.Element("GrandChild");

    foreach (string str in grandChildData) {
        Console.WriteLine(str);
    }
}
```

Este ejemplo produce el siguiente resultado:

```
bbb
ccc
```

En este ejemplo el documento de origen es muy pequeño. No obstante, aunque hubiera millones de elementos `Child`, este ejemplo seguiría teniendo una superficie de memoria pequeña.

Procedimiento para rellenar un árbol XML con un objeto XmlWriter (LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Una forma de rellenar un árbol XML consiste en utilizar CreateWriter para crear XmlWriter y después escribir en XmlWriter. El árbol XML se rellena con todos los nodos que se escriben en XmlWriter.

Normalmente se usa este método cuando se usa LINQ to XML con otra clase que espera escribir en un XmlWriter, como XslCompiledTransform.

Ejemplo

Un uso posible para CreateWriter es la invocación de una transformación XSLT. Este ejemplo crea un árbol XML, crea un XmlReader del árbol XML, crea un nuevo documento y después crea un XmlWriter para escribir en el nuevo documento. A continuación invoca la transformación XSLT, pasando XmlReader y XmlWriter. Después de que se complete correctamente la transformación, se rellenará el nuevo árbol XML con los resultados de la transformación.

```
string xslMarkup = @"<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>
    <xsl:template match='/Parent'>
        <Root>
            <C1>
            <xsl:value-of select='Child1'/>
            </C1>
            <C2>
            <xsl:value-of select='Child2'/>
            </C2>
        </Root>
    </xsl:template>
</xsl:stylesheet>";

XDocument xmlTree = new XDocument(
    new XElement("Parent",
        new XElement("Child1", "Child1 data"),
        new XElement("Child2", "Child2 data")
    )
);

XDocument newTree = new XDocument();
using (XmlWriter writer = newTree.CreateWriter())
{
    // Load the style sheet.
    XslCompiledTransform xslt = new XslCompiledTransform();
    xslt.Load(XmlReader.Create(new StringReader(xslMarkup)));

    // Execute the transformation and output the results to a writer.
    xslt.Transform(xmlTree.CreateReader(), writer);
}

Console.WriteLine(newTree);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <C1>Child1 data</C1>
  <C2>Child2 data</C2>
</Root>
```

Vea también

- CreateWriter
- XmlWriter
- XslCompiledTransform
- Crear árboles XML (C#)

Procedimiento para realizar validaciones con XSD (LINQ to XML) (C#)

18/03/2020 • 3 minutes to read • Edit Online

El espacio de nombres System.Xml.Schema contiene métodos de extensión que hacen que sea fácil validar un árbol XML contra un archivo de lenguaje de definición de esquemas XML (XSD). Para obtener más información, vea la documentación del método Validate.

Ejemplo

En el ejemplo siguiente se crea un XmlSchemaSet, a continuación, se validan dos objetos XDocument con el esquema establecido. Uno de los documentos es válido, el otro no.

```
string xsdMarkup =
    @"<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
       <xsd:element name='Root'>
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element name='Child1' minOccurs='1' maxOccurs='1'/>
          <xsd:element name='Child2' minOccurs='1' maxOccurs='1'/>
         </xsd:sequence>
        </xsd:complexType>
       </xsd:element>
      </xsd:schema>";
XmlSchemaSet schemas = new XmlSchemaSet();
schemas.Add("", XmlReader.Create(new StringReader(xsdMarkup)));

XDocument doc1 = new XDocument(
    new XElement("Root",
        new XElement("Child1", "content1"),
        new XElement("Child2", "content1")
    )
);

XDocument doc2 = new XDocument(
    new XElement("Root",
        new XElement("Child1", "content1"),
        new XElement("Child3", "content1")
    )
);

Console.WriteLine("Validating doc1");
bool errors = false;
doc1.Validate(schemas, (o, e) =>
                    {
                        Console.WriteLine("{0}", e.Message);
                        errors = true;
                    });
Console.WriteLine("doc1 {0}", errors ? "did not validate" : "validated");

Console.WriteLine();
Console.WriteLine("Validating doc2");
errors = false;
doc2.Validate(schemas, (o, e) =>
                    {
                        Console.WriteLine("{0}", e.Message);
                        errors = true;
                    });
Console.WriteLine("doc2 {0}", errors ? "did not validate" : "validated");
```

Este ejemplo produce el siguiente resultado:

```
Validating doc1
doc1 validated

Validating doc2
The element 'Root' has invalid child element 'Child3'. List of possible elements expected: 'Child2'.
doc2 did not validate
```

Ejemplo

El siguiente ejemplo valida que el documento XML de Archivo XML de muestra: clientes y pedidos (LINQ to XML) sea válido según el esquema de Archivo XSD de muestra: clientes y pedidos. A continuación modifica el documento XML de origen. Cambia el atributo `CustomerID` del primer cliente. Tras el cambio, los pedidos harán referencia a un cliente que no existe, de forma que el documento XML ya no se validará.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

En este ejemplo se usa el siguiente esquema XSD: Sample XSD File: Customers and Orders (Archivo XSD de ejemplo: Clientes y pedidos).

```
XmlSchemaSet schemas = new XmlSchemaSet();
schemas.Add("", "CustomersOrders.xsd");

Console.WriteLine("Attempting to validate");
XDocument custOrdDoc = XDocument.Load("CustomersOrders.xml");
bool errors = false;
custOrdDoc.Validate(schemas, (o, e) =>
                    {
                        Console.WriteLine("{0}", e.Message);
                        errors = true;
                    });
Console.WriteLine("custOrdDoc {0}", errors ? "did not validate" : "validated");

Console.WriteLine();
// Modify the source document so that it will not validate.
custOrdDoc.Root.Element("Orders").Element("Order").Element("CustomerID").Value = "AAAAA";
Console.WriteLine("Attempting to validate after modification");
errors = false;
custOrdDoc.Validate(schemas, (o, e) =>
                    {
                        Console.WriteLine("{0}", e.Message);
                        errors = true;
                    });
Console.WriteLine("custOrdDoc {0}", errors ? "did not validate" : "validated");
```

Este ejemplo produce el siguiente resultado:

```
Attempting to validate
custOrdDoc validated

Attempting to validate after modification
The key sequence 'AAAAA' in Keyref fails to refer to some key.
custOrdDoc did not validate
```

Vea también

- Validate
- Crear árboles XML (C#)

Contenido válido de objetos XElement y XDocument

18/03/2020 • 5 minutes to read • Edit Online

En este tema se describen los argumentos válidos que se pueden pasar a los constructores y los métodos que se usan para agregar contenido a elementos y documentos.

Contenido válido

Las consultas a menudo evalúan a IEnumerable<T> de XElement o a IEnumerable<T> de XAttribute. Se pueden pasar colecciones de objetos XElement o XAttribute al constructor XElement. Por lo tanto, resulta conveniente pasar los resultados de una consulta como contenido a los métodos y constructores que use para rellenar árboles XML.

Al agregar contenido simple, se pueden pasar varios tipos a este método. Entre los tipos válidos se incluyen los siguientes:

- String
- Double
- Single
- Decimal
- Boolean
- DateTime
- TimeSpan
- DateTimeOffset
- Cualquier tipo que implemente `Object.ToString`.
- Cualquier tipo que implemente IEnumerable<T>.

Al agregar contenido complejo, se pueden pasar varios tipos a este método:

- XObject
- XNode
- XAttribute
- Cualquier tipo que implemente IEnumerable<T>

Si un objeto implementa IEnumerable<T>, se enumera la colección del objeto y se agregan todos los elementos de la colección. Si la colección contiene objetos XNode o XAttribute, cada elemento de la colección se agrega por separado. Si la colección contiene texto (u objetos convertidos a texto), el texto de la colección se concatena y se agrega como un nodo de texto.

Si el contenido es `null`, no se agrega nada. Al pasar una colección, se permite que los elementos de la colección sean `null`. Un elemento `null` de la colección no tiene ningún efecto en el árbol.

Un atributo agregado debe tener un nombre único en el elemento contenedor.

Cuando se asocian objetos XNode o XAttribute, si el contenido nuevo no tiene un elemento primario, los objetos simplemente se adjuntan al árbol XML. Si el contenido nuevo ya tiene un elemento primario y forma parte de otro

árbol XML, el nuevo contenido se clonará y dicho clon se adjuntará al árbol XML.

Contenido válido para documentos

Los atributos y el contenido simple no se pueden agregar a un documento.

No existen muchos escenarios que requieran la creación de un objeto XDocument. En su lugar, normalmente puede crear los árboles XML con un nodo raíz XElement. A menos que exista un requisito específico de crear un documento (por ejemplo, porque deba crear instrucciones y comentarios de procesamiento en el nivel superior, o bien deba admitir tipos de documento), a menudo resulta más conveniente usar XElement como nodo raíz.

El contenido válido para un documento incluye lo siguiente:

- Cero o un objeto XDocumentType. Los tipos de documento deben ir antes del elemento.
- Cero o un elemento.
- Cero o más comentarios.
- Cero o más instrucciones de procesamiento.
- Cero o más nodos de texto que contengan solo espacios en blanco.

Constructores y funciones que permiten agregar contenido

Los métodos siguientes permiten agregar contenido secundario a un objeto XElement o a un objeto XDocument:

MÉTODO	DESCRIPTION
XElement	Construye un objeto XElement.
XDocument	Construye un objeto XDocument.
Add	Agrega al final del contenido secundario del objeto XElement o del objeto XDocument.
AddAfterSelf	Añade un contenido detrás de XNode.
AddBeforeSelf	Agrega contenido antes de XNode.
AddFirst	Agrega un contenido al comienzo de los contenidos secundarios del XContainer.
ReplaceAll	Reemplaza todo el contenido (atributos y nodos secundarios) de un objeto XElement.
ReplaceAttributes	Reemplaza los atributos de un objeto XElement.
ReplaceNodes	Reemplaza los nodos secundarios por contenido nuevo.
ReplaceWith	Reemplaza un nodo por contenido nuevo.

Vea también

- Crear árboles XML (C#)

Información general sobre los espacios de nombres (LINQ to XML)

29/06/2020 • 3 minutes to read • Edit Online

En este artículo se presentan los espacios de nombres, la clase XName y la clase XNamespace.

Nombres XML

Los nombres XML a menudo son una fuente de complejidad en la programación XML. Un nombre XML se compone de un espacio de nombres XML (también denominado URI de espacio de nombres XML) y un nombre local. Un espacio de nombres XML es similar a un espacio de nombres en un programa de .NET. Permite calificar de manera exclusiva los nombres de los elementos y los atributos. Evita conflictos de nombres entre varias partes de un documento XML. Una vez que haya declarado un espacio de nombres XML, puede seleccionar un nombre local que sólo debe ser único en dicho espacio de nombres.

Otro aspecto de los nombres XML son los *prefijos de espacios de nombres* XML. Los prefijos XML generan la mayor parte de la complejidad de los nombres XML. Estos prefijos permiten crear un acceso directo de un espacio de nombres XML, lo que hace que el documento XML sea más conciso y comprensible. Sin embargo, los prefijos XML dependen de su contexto para tener un significado, lo que aumenta la complejidad. Por ejemplo, el prefijo XML `aw` podría asociarse a un espacio de nombres XML en una parte del árbol XML y a un espacio de nombres XML diferente en una parte distinta de dicho árbol.

Una de la ventajas de usar LINQ to XML con C# consiste en no tener que usar prefijos XML. Cuando LINQ to XML carga o analiza un documento XML, cada prefijo XML se resuelve en su espacio de nombres XML correspondiente. Después, cuando trabaje con un documento que usa espacios de nombres, la mayoría de las veces tendrá acceso a dichos espacios de nombres mediante el URI de éstos, y no mediante prefijo. Cuando los desarrolladores trabajan con nombres XML en LINQ to XML, siempre lo hacen con un nombre XML completo (es decir, un espacio de nombres XML y un nombre local). Sin embargo, si es necesario, LINQ to XML permite trabajar con prefijos de espacios de nombres y controlarlos.

En LINQ to XML, la clase que representa los nombres XML es XName. Los nombres XML aparecen frecuentemente en la API LINQ to XML, y cuando se requiera un nombre XML, encontrará un parámetro XName. En cambio, apenas se trabaja directamente con un objeto XName. XName contiene una conversión implícita de cadena.

Para obtener más información, vea XNamespace y XName.

Procedimiento para crear un documento con espacios de nombres (C#) (LINQ to XML)

18/03/2020 • 5 minutes to read • Edit Online

En este tema se muestra cómo crear documentos con espacios de nombres.

Ejemplo

Para crear un elemento o un atributo que se encuentra en un espacio de nombres, primero debe declarar e inicializar un objeto XNamespace. A continuación debe utilizar la sobrecarga del operador de suma para combinar el espacio de nombres con el nombre local, expresado como una cadena.

En el ejemplo siguiente se crea un documento con un espacio de nombres. De forma predeterminada, LINQ to XML serializa este documento con un espacio de nombres predeterminado.

```
// Create an XML tree in a namespace.
XNamespace aw = "http://www.adventure-works.com";
XElement root = new XElement(aw + "Root",
    new XElement(aw + "Child", "child content")
);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<Root xmlns="http://www.adventure-works.com">
  <Child>child content</Child>
</Root>
```

Ejemplo

En el ejemplo siguiente se crea un documento con un espacio de nombres. También crea un atributo que declara el espacio de nombres con un prefijo. Para crear un atributo que declara un espacio de nombres con un prefijo, debe crearlo de modo que el nombre del atributo sea el prefijo del espacio de nombres y el nombre se encuentre en el espacio de nombres Xmlns. El valor de este atributo es el URI del espacio de nombres.

```
// Create an XML tree in a namespace, with a specified prefix
XNamespace aw = "http://www.adventure-works.com";
XElement root = new XElement(aw + "Root",
    new XAttribute(XNamespace.Xmlns + "aw", "http://www.adventure-works.com"),
    new XElement(aw + "Child", "child content")
);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<aw:Root xmlns:aw="http://www.adventure-works.com">
  <aw:Child>child content</aw:Child>
</aw:Root>
```

Ejemplo

El ejemplo siguiente muestra la creación de un documento que contiene dos espacios de nombres. Uno es el espacio de nombres predeterminado. El otro es un espacio de nombres con un prefijo.

Mediante la inclusión de los atributos de espacio de nombres en el elemento raíz, los espacios de nombres se serializan de modo que `http://www.adventure-works.com` sea el espacio de nombres predeterminado y `www.fourthcoffee.com` se serialice con un prefijo "fc". Para crear un atributo que declare un espacio de nombres predeterminado, debe crear un atributo con el nombre "xmlns", sin un espacio de nombres. El valor del atributo es el URI del espacio de nombres predeterminado.

```
// The http://www.adventure-works.com namespace is forced to be the default namespace.
XNamespace aw = "http://www.adventure-works.com";
XNamespace fc = "www.fourthcoffee.com";
XElement root = new XElement(aw + "Root",
    new XAttribute("xmlns", "http://www.adventure-works.com"),
    new XAttribute(XNamespace.Xmlns + "fc", "www.fourthcoffee.com"),
    new XElement(fc + "Child",
        new XElement(aw + "DifferentChild", "other content")
    ),
    new XElement(aw + "Child2", "c2 content"),
    new XElement(fc + "Child3", "c3 content")
);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<Root xmlns="http://www.adventure-works.com" xmlns:fc="www.fourthcoffee.com">
  <fc:Child>
    <DifferentChild>other content</DifferentChild>
  </fc:Child>
  <Child2>c2 content</Child2>
  <fc:Child3>c3 content</fc:Child3>
</Root>
```

Ejemplo

En el ejemplo siguiente se crea un documento que contiene dos espacios de nombres, todos ellos con prefijos de espacio de nombres.

```
XNamespace aw = "http://www.adventure-works.com";
XNamespace fc = "www.fourthcoffee.com";
XElement root = new XElement(aw + "Root",
    new XAttribute(XNamespace.Xmlns + "aw", aw.NamespaceName),
    new XAttribute(XNamespace.Xmlns + "fc", fc.NamespaceName),
    new XElement(fc + "Child",
        new XElement(aw + "DifferentChild", "other content")
    ),
    new XElement(aw + "Child2", "c2 content"),
    new XElement(fc + "Child3", "c3 content")
);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<aw:Root xmlns:aw="http://www.adventure-works.com" xmlns:fc="www.fourthcoffee.com">
  <fc:Child>
    <aw:DifferentChild>other content</aw:DifferentChild>
  </fc:Child>
  <aw:Child2>c2 content</aw:Child2>
  <fc:Child3>c3 content</fc:Child3>
</aw:Root>
```

Ejemplo

Otro método para conseguir el mismo resultado consiste en usar nombres expandidos en lugar de declarar y crear un objeto XNamespace.

Este método tiene implicaciones en el rendimiento. Cada vez que pasa una cadena que contiene un nombre expandido a LINQ to XML, LINQ to XML debe analizar el nombre, buscar el espacio de nombres atomizado y buscar el nombre atomizado. Este proceso consume tiempo de la CPU. Si el rendimiento es importante, tal vez quiera declarar y usar explícitamente un objeto XNamespace.

Si el rendimiento es un problema importante, vea Atomización previa de objetos XName (LINQ to XML) (C#) para obtener más información.

```
// Create an XML tree in a namespace, with a specified prefix
XElement root = new XElement("{http://www.adventure-works.com}Root",
    new XAttribute(XNamespace.Xmlns + "aw", "http://www.adventure-works.com"),
    new XElement("{http://www.adventure-works.com}Child", "child content")
);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<aw:Root xmlns:aw="http://www.adventure-works.com">
  <aw:Child>child content</aw:Child>
</aw:Root>
```

Vea también

- Información general sobre los espacios de nombres (LINQ to XML) (C#)

Procedimiento para controlar prefijos de espacios de nombres (C#) (LINQ to XML)

18/03/2020 • 3 minutes to read • Edit Online

En este tema se describe cómo puede controlar los prefijos de espacios de nombres al serializar un árbol XML.

En muchas situaciones, no es necesario controlar los prefijos de espacios de nombres.

Sin embargo, ciertas herramientas de programación XML requieren un control específico de los prefijos de espacios de nombres. Por ejemplo, es posible que esté manipulando una hoja de estilos XSLT o un documento XAML que contenga expresiones XPath incrustadas que hagan referencia a prefijos de espacios de nombres; en ese caso, es importante serializar el documento con dichos prefijos.

Esta es la razón más común para controlar los prefijos de espacio de nombres.

Otro motivo habitual para controlar prefijos de espacios de nombres se basa en permitir que los usuarios editen el documento XML manualmente y crear prefijos de fácil escritura. Por ejemplo, tal vez esté generando un documento XSD. Las convenciones de los esquemas sugieren el uso de `xs` o `xsd` como prefijo para el espacio de nombres de esquema.

Para controlar los prefijos de espacios de nombre, inserta atributos que declaran dichos espacios de nombres. Si declara los espacios de nombres con prefijos específicos, LINQ to XML intentará respetar los prefijos durante la serialización.

Para crear un atributo que declara un espacio de nombres con un prefijo, puede crear un atributo donde el espacio de nombres del nombre del atributo es Xmlns y el nombre del atributo es el prefijo del espacio de nombres. El valor del atributo es el URI del espacio de nombres.

Ejemplo

Este ejemplo declara dos espacios de nombres. Especifica que el espacio de nombres `http://www.adventure-works.com` tiene el prefijo `aw` y que el espacio de nombres `www.fourthcoffee.com` tiene el prefijo `fc`.

```
XNamespace aw = "http://www.adventure-works.com";
XNamespace fc = "www.fourthcoffee.com";
XElement root = new XElement(aw + "Root",
    new XAttribute(XNamespace.Xmlns + "aw", "http://www.adventure-works.com"),
    new XAttribute(XNamespace.Xmlns + "fc", "www.fourthcoffee.com"),
    new XElement(fc + "Child",
        new XElement(aw + "DifferentChild", "other content")
    ),
    new XElement(aw + "Child2", "c2 content"),
    new XElement(fc + "Child3", "c3 content")
);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<aw:Root sinlns:aw="http://www.adventure-works.com" sinlns:fc="www.fourthcoffee.com">
  <fc:Child>
    <aw:DifferentChild>other content</aw:DifferentChild>
  </fc:Child>
  <aw:Child2>c2 content</aw:Child2>
  <fc:Child3>c3 content</fc:Child3>
</aw:Root>
```

Vea también

- Información general sobre los espacios de nombres (LINQ to XML) (C#)

Ámbito del espacio de nombres predeterminado de C#

18/03/2020 • 2 minutes to read • Edit Online

Los espacios de nombres predeterminados del árbol XML no se encuentran en el ámbito de las consultas. Si tiene código XML en un espacio de nombres predeterminado, debe declarar una variable XNamespace y combinarla con el nombre local para convertirla en un nombre completo que se usará en la consulta.

Uno de los problemas más habituales al consultar árboles XML es que si el árbol XML tiene un espacio de nombres predeterminado, el desarrollador a veces escribe la consulta como si el código XML no estuviera en un espacio de nombres.

El primer conjunto de ejemplos de este tema muestra una forma habitual de cargar XML en un espacio de nombres predeterminado con una consulta incorrecta.

El segundo conjunto de ejemplos muestra las correcciones necesarias para que pueda consultar el código XML en un espacio de nombres.

Ejemplo

En este ejemplo se muestra la creación de XML en un espacio de nombres y una consulta que devuelve un conjunto de resultados vacío.

Código

```
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
    <Child>1</Child>
    <Child>2</Child>
    <Child>3</Child>
    <AnotherChild>4</AnotherChild>
    <AnotherChild>5</AnotherChild>
    <AnotherChild>6</AnotherChild>
</Root>");
IEnumerable<XElement> c1 =
    from el in root.Elements("Child")
    select el;
Console.WriteLine("Result set follows:");
foreach (XElement el in c1)
    Console.WriteLine((int)el);
Console.WriteLine("End of result set");
```

Comentarios

Este ejemplo genera el siguiente resultado:

```
Result set follows:
End of result set
```

Ejemplo

En este ejemplo se muestra la creación de XML en un espacio de nombres y una consulta que se codifica correctamente.

A diferencia del anterior ejemplo de código incorrecto, el enfoque adecuado al usar C# consiste en declarar e inicializar un objeto XNamespace y usarlo cuando se especifiquen objetos XName. En este caso, el argumento para el método Elements es un objeto XName.

Código

```
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
    <Child>1</Child>
    <Child>2</Child>
    <Child>3</Child>
    <AnotherChild>4</AnotherChild>
    <AnotherChild>5</AnotherChild>
    <AnotherChild>6</AnotherChild>
</Root>");
XNamespace aw = "http://www.adventure-works.com";
IEnumerable<XElement> c1 =
    from el in root.Elements(aw + "Child")
    select el;
Console.WriteLine("Result set follows:");
foreach (XElement el in c1)
    Console.WriteLine((int)el);
Console.WriteLine("End of result set");
```

Comentarios

Este ejemplo genera el siguiente resultado:

```
Result set follows:
1
2
3
End of result set
```

Vea también

- Información general sobre los espacios de nombres (LINQ to XML) (C#)

Procedimiento para escribir consultas de XML en espacios de nombres (C#)

18/03/2020 • 2 minutes to read • Edit Online

Para escribir una consulta en XML que esté en un espacio de nombres, debe usar objetos XName que tengan el espacio de nombres correcto.

Para C#, el enfoque más común consiste en inicializar un objeto XNamespace usando una cadena que contiene el identificador URI y, después, usar la sobrecarga del operador de suma para combinar el espacio de nombres con el nombre local.

En el primer conjunto de ejemplos de este tema se muestra cómo crear un árbol XML en un espacio de nombres predeterminado. En el segundo conjunto se muestra cómo crear un árbol XML en un espacio de nombres con un prefijo.

Ejemplo

En el ejemplo siguiente se crea un árbol XML que está en un espacio de nombres predeterminado. A continuación recupera una recopilación de elementos.

```
XNamespace aw = "http://www.adventure-works.com";
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
    <Child>1</Child>
    <Child>2</Child>
    <Child>3</Child>
    <AnotherChild>4</AnotherChild>
    <AnotherChild>5</AnotherChild>
    <AnotherChild>6</AnotherChild>
</Root>");
IEnumerable<XElement> c1 =
    from el in root.Elements(aw + "Child")
    select el;
foreach (XElement el in c1)
    Console.WriteLine((int)el);
```

Este ejemplo produce el siguiente resultado:

```
1
2
3
```

Ejemplo

En C#, las consultas se escribe de la misma forma, independientemente de si se escribe en un árbol XML que usa un espacio de nombres con un prefijo o en un árbol XML con un espacio de nombres predeterminado.

En el ejemplo siguiente se crea un árbol XML que está en un espacio de nombres con un prefijo. A continuación recupera una recopilación de elementos.

```
XNamespace aw = "http://www.adventure-works.com";
XElement root = XElement.Parse(
@"<aw:Root xmlns:aw='http://www.adventure-works.com'>
    <aw:Child>1</aw:Child>
    <aw:Child>2</aw:Child>
    <aw:Child>3</aw:Child>
    <aw:AnotherChild>4</aw:AnotherChild>
    <aw:AnotherChild>5</aw:AnotherChild>
    <aw:AnotherChild>6</aw:AnotherChild>
</aw:Root>");
IEnumerable<XElement> c1 =
    from el in root.Elements(aw + "Child")
    select el;
foreach (XElement el in c1)
    Console.WriteLine((int)el);
```

Este ejemplo produce el siguiente resultado:

```
1
2
3
```

Vea también

- Información general sobre los espacios de nombres (LINQ to XML) (C#)

Mantener un espacio en blanco al serializar

18/03/2020 • 2 minutes to read • Edit Online

En este tema se describe cómo controlar los espacios en blanco a la hora de serializar un árbol XML.

Un caso muy común es aquel en el que se leen datos XML con sangría, se crea un árbol XML en memoria sin ningún nodo de texto con espacios en blanco (es decir, sin conservar los espacios en blanco), se realizan ciertas operaciones sobre el XML y este se guarda con sangría. Si se serializa el XML con formato, solo se preservan en el XML aquellos espacios en blanco más significativos. Éste es el comportamiento predeterminado en LINQ to XML.

Otro escenario muy común es aquel en el que se lee y se modifica código XML en el que se ha aplicado sangría de forma intencionada. Es posible que no desee modificar esta sangría de ninguna forma. Para hacerlo en LINQ to XML, puede preservar los espacios en blanco a la hora de cargar o analizar el XML y si deshabilita el formato cuando serialice el XML.

Tratamiento de los espacios en blanco en los métodos que serializan árboles XML

Los siguientes métodos de las clases XElement y XDocument serializan un árbol XML. Es posible serializar un árbol XML en un archivo, un TextReader o un XmlReader. El método `ToString` serializa en una cadena.

- XElement.Save
- XDocument.Save
- XElement.ToString()
- XDocument.ToString()

Si el método no recibe un SaveOptions como argumento, entonces el método realizará un formato (sangría) del XML serializado. En ese caso, se descartarán todos los espacios en blanco del árbol XML que no sean significativos.

Si el método toma SaveOptions como argumento, se puede especificar que el método no dé formato (sangría) al XML serializado. En ese caso, se preservarán todos los espacios en blanco del árbol XML.

Serializar con una declaración XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se describe cómo controlar si la serialización genera una declaración XML.

Generación de declaración XML

La serialización en File o TextWriter mediante el método XElement.Save o el método XDocument.Save genera una declaración XML. Cuando serializa en XmlWriter, la configuración del escritor (especificada en un objeto XmlWriterSettings) determina si se genera una declaración XML.

Si está serializando en una cadena mediante el método `ToString`, el XML resultante no incluirá una declaración XML.

Serializar con una declaración XML

En el ejemplo siguiente se crea un XElement, se guarda el documento en un archivo y, a continuación, se imprime el archivo en la consola:

```
XElement root = new XElement("Root",
    new XElement("Child", "child content")
);
root.Save("Root.xml");
string str = File.ReadAllText("Root.xml");
Console.WriteLine(str);
```

Este ejemplo produce el siguiente resultado:

```
<?xml version="1.0" encoding="utf-8"?>
<Root>
  <Child>child content</Child>
</Root>
```

Serializar sin una declaración XML

En el siguiente ejemplo se muestra cómo guardar un XElement en un XmlWriter.

```
StringBuilder sb = new StringBuilder();
XmlWriterSettings xws = new XmlWriterSettings();
xws.OmitXmlDeclaration = true;

using (XmlWriter xw = XmlWriter.Create(sb, xws)) {
    XElement root = new XElement("Root",
        new XElement("Child", "child content")
    );
    root.Save(xw);
}
Console.WriteLine(sb.ToString());
```

Este ejemplo produce el siguiente resultado:

```
<Root><Child>child content</Child></Root>
```

Vea también

- Serializar árboles XML (C#)

Serializar en archivos, escritores de texto y escritores XML

18/03/2020 • 2 minutes to read • Edit Online

Puede serializar árboles XML en File, TextWriter o XmlWriter.

Puede serializar cualquier componente XML, entre los que se incluyen XDocument y XElement, en una cadena con el método `ToString`.

Si desea suprimir el formato al serializar en una cadena, puede usar el método XNode.ToString.

El comportamiento predeterminado al serializar en un archivo consiste en formatear el documento XML resultante (aplicarle una sangría). Cuando aplica una sangría, en el árbol XML no se conservan los espacios en blanco más insignificantes. Para serializar con formato, use una de las sobrecargas de los métodos siguientes que no toman SaveOptions como argumento:

- XDocument.Save
- XElement.Save

Si desea que la opción no aplique una sangría y conserve los espacios en blanco más insignificantes del árbol XML, use una de las sobrecargas de los métodos siguientes que toman SaveOptions como argumento:

- XDocument.Save
- XElement.Save

Por obtener ejemplos, vea el tema de referencia correspondiente.

Vea también

- Serializar árboles XML (C#)

Serializar en un objeto XmlReader (invocando XSLT) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Cuando use las funciones de interoperabilidad System.Xml de LINQ to XML, puede usar CreateReader para crear un objeto XmlReader. El módulo que lee desde el objeto XmlReader lee los nodos desde el árbol XML y los procesa.

Invocar una transformación XSLT

Un uso posible para este método es la invocación de una transformación XSLT. Puede crear un árbol XML, crear un objeto XmlReader desde el árbol XML, crear un nuevo documento y crear un objeto XmlWriter que escribirá en el nuevo documento. A continuación, puede invocar la transformación XSLT, pasando XmlReader y XmlWriter. Después de que se complete correctamente la transformación, se rellenará el nuevo árbol XML con los resultados de la transformación.

```
string xslMarkup = @"<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>
    <xsl:template match='/Parent'>
        <Root>
            <C1>
            <xsl:value-of select='Child1'/>
            </C1>
            <C2>
            <xsl:value-of select='Child2'/>
            </C2>
        </Root>
    </xsl:template>
</xsl:stylesheet>";

XDocument xmlTree = new XDocument(
    new XElement("Parent",
        new XElement("Child1", "Child1 data"),
        new XElement("Child2", "Child2 data")
    )
);

XDocument newTree = new XDocument();
using (XmlWriter writer = newTree.CreateWriter()) {
    // Load the style sheet.
    XslCompiledTransform xslt = new XslCompiledTransform();
    xslt.Load(XmlReader.Create(new StringReader(xslMarkup)));

    // Execute the transformation and output the results to a writer.
    xslt.Transform(xmlTree.CreateReader(), writer);
}

Console.WriteLine(newTree);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <C1>Child1 data</C1>
  <C2>Child2 data</C2>
</Root>
```

Vea también

- Serializar árboles XML (C#)

Información general sobre los ejes de LINQ to XML (C#)

18/03/2020 • 7 minutes to read • Edit Online

Tras crear un árbol XML o haber cargado un documento XML en un árbol XML, puede realizar consultas sobre él para encontrar elementos y atributos, así como para recuperar sus valores. Las colecciones se recuperan mediante los *métodos de eje*, también denominados *ejes*. Algunos de los ejes son métodos de las clases XElement y XDocument, que devuelven colecciones IEnumerable<T>. Algunos de los métodos Axes son métodos de extensión de la clase Extensions. Los ejes que se han implementado como métodos de extensión trabajan colecciones y devuelven colecciones.

Como se describe en Información general acerca de la clase XElement, un objeto XElement representa un nodo de elemento único. El contenido de un elemento puede ser complejo (a veces conocido como contenido estructurado) o puede ser un elemento simple. Un elemento simple puede estar vacío o contener un valor. Si el nodo incluye contenido estructurado, puede utilizar diferentes métodos del eje para recuperar enumeraciones con los elementos descendientes. Los métodos de ejes que se utilizan con más frecuencia son Elements y Descendants.

Además de los métodos de eje, que devuelven colecciones, existen otros dos métodos que usará con regularidad en consultas LINQ to XML. El método Element devuelve un único XElement. El método Attribute devuelve un único XAttribute.

Por diversas razones, las consultas LINK ofrecen el mecanismo más potente para examinar árboles, extraer datos de ellos y transformarlos. Las consultas LINQ operan sobre objetos que implementan IEnumerable<T>, y los ejes LINQ to XML devuelven IEnumerable<T> de colecciones XElement, y IEnumerable<T> de colecciones XAttribute. Para poder realizar las consultas, necesitará estas colecciones.

A parte de los métodos de ejes que permiten recuperar colecciones de elementos y atributos, existen métodos de ejes que le permiten iterar por un árbol con gran detalle. Por ejemplo, en vez de tratar directamente con los elementos y los atributos, podría trabajar con los nodos del árbol. Los nodos tienen un nivel de granularidad mayor que los elementos y atributos. Si trabaja con nodos, podrá examinar comentarios XML, nodos de texto, instrucciones de procesamiento, etc. Esta funcionalidad resultaría interesante, por ejemplo, para una persona que estuviese escribiendo un procesador de textos y quisiera guardar los documentos en formato XML. No obstante, la mayoría de programadores en XML se centran más en los elementos, los atributos y sus valores.

Métodos para recuperar una colección de elementos

A continuación encontrará un resumen de los métodos de la clase XElement (o de sus clases base) a los que podrá pasar un XElement para devolver una colección de elementos.

MÉTODO	DESCRIPTION
XNode.Ancestors	Devuelve un IEnumerable<T> de XElement de los antecesores de este elemento. Una sobrecarga devuelve un IEnumerable<T> de XElement de los antecesores que tengan el XName especificado.

MÉTODO	DESCRIPTION
XContainer.Descendants	Devuelve un IEnumerable<T> de XElement de los descendientes de este elemento. Una sobrecarga devuelve un IEnumerable<T> de XElement de los descendientes que tengan el XName especificado.
XContainer.Elements	Devuelve un IEnumerable<T> de XElement de los elementos secundarios dependientes de este elemento. Una sobrecarga devuelve un IEnumerable<T> de XElement de los elementos secundarios que tengan el XName especificado.
XNode.ElementsAfterSelf	Devuelve un IEnumerable<T> de XElement de los elementos que se encuentran después de este elemento. Una sobrecarga devuelve un IEnumerable<T> de XElement de los elementos que se encuentren después de este elemento y que tengan el XName especificado.
XNode.ElementsBeforeSelf	Devuelve un IEnumerable<T> de XElement de los elementos que se encuentran antes que este elemento. Una sobrecarga devuelve un IEnumerable<T> de XElement de los elementos que se encuentren antes que este elemento y que tengan el XName especificado.
XElement.AncestorsAndSelf	Devuelve un IEnumerable<T> de XElement de este elemento y de sus antecesores. Una sobrecarga devuelve un IEnumerable<T> de XElement de los elementos que tengan el XName especificado.
XElement.DescendantsAndSelf	Devuelve un IEnumerable<T> de XElement de este elemento y de sus descendientes. Una sobrecarga devuelve un IEnumerable<T> de XElement de los elementos que tengan el XName especificado.

Método para recuperar un único elemento

El siguiente método recupera un único elemento secundario a partir del objeto XElement.

MÉTODO	DESCRIPTION
XContainer.Element	Devuelve el objeto del primer elemento XElement secundario que tenga el XName especificado.

Método para recuperar una colección de atributos

El siguiente método recupera atributos a partir de un objeto XElement.

MÉTODO	DESCRIPTION
XElement.Attributes	Devuelve un IEnumerable<T> de XAttribute de todos los atributos.

Método para recuperar un único atributo

El siguiente método recupera un único atributo a partir de un objeto XElement.

MÉTODO	DESCRIPTION
XElement.Attribute	Devuelve el XAttribute que tenga el XName especificado.

Vea también

- LINQ to XML Axes (C#) (Ejes de LINQ to XML [C#])

Procedimiento para recuperar una colección de elementos (LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se demuestra el método Elements. Este método recupera una colección de elementos secundarios de un elemento.

Ejemplo

Este ejemplo recorre en iteración los elementos secundarios del elemento `purchaseOrder`.

En este ejemplo se usa el siguiente documento XML: Sample XML File: Typical Purchase Order (LINQ to XML) (Archivo XML de ejemplo: pedido de compra común [LINQ to XML]).

```
XElement po = XElement.Load("PurchaseOrder.xml");
IEnumerable<XElement> childElements =
    from el in po.Elements()
    select el;
foreach (XElement el in childElements)
    Console.WriteLine("Name: " + el.Name);
```

Este ejemplo produce el siguiente resultado:

```
Name: Address
Name: Address
Name: DeliveryNotes
Name: Items
```

Vea también

- LINQ to XML Axes (C#) (Ejes de LINQ to XML [C#])

Procedimiento para recuperar el valor de un elemento (LINQ to XML) (C#)

16/04/2020 • 5 minutes to read • Edit Online

En este artículo se muestra cómo obtener el valor de los elementos. Existen dos formas principales de hacerlo:

- Convertir un objeto XElement o un objeto XAttribute al tipo deseado. El operador de conversión explícita convierte el contenido del elemento o del atributo al tipo especificado y lo asigna a la variable.
- Use las propiedades XElement.Value y XAttribute.Value. También puede establecer el valor con estas propiedades.

Con C#, la conversión suele ser el mejor método. Si convierte el elemento o el atributo a un tipo que admite valores NULL, resulta más fácil escribir el código al recuperar el valor de un elemento (o atributo) que podría existir o no. En el último ejemplo de este artículo se muestra que la conversión es más sencilla en caso de que el elemento no exista. No obstante, no puede establecer el contenido de un elemento mediante la conversión del mismo modo que con la propiedad XElement.Value.

Ejemplo de conversión de cadena

Para recuperar el valor de un elemento, convierta el objeto XElement al tipo deseado. Siempre puede convertir un elemento en cadena, tal y como se indica a continuación:

```
XElement e = new XElement("StringElement", "abcde");
Console.WriteLine(e);
Console.WriteLine("Value of e:" + (string)e);
```

Este ejemplo produce el siguiente resultado:

```
<StringElement>abcde</StringElement>
Value of e:abcde
```

Ejemplo de conversión de enteros

También puede convertir elementos a tipos diferentes a una cadena. Por ejemplo, si tiene un elemento que contiene un entero, puede convertirlo a `int`, tal como se muestra en el código siguiente:

```
XElement e = new XElement("Age", "44");
Console.WriteLine(e);
Console.WriteLine("Value of e:" + (int)e);
```

Este ejemplo produce el siguiente resultado:

```
<Age>44</Age>
Value of e:44
```

LINQ to XML proporciona operadores de conversión explícita para los tipos de datos siguientes: `string`, `bool`, `bool?`, `int`, `int?`, `uint`, `uint?`, `long`, `long?`, `ulong`, `ulong?`, `float`, `float?`, `double`, `double?`, `decimal`,

`decimal?`, `DateTime`, `DateTime?`, `TimeSpan`, `TimeSpan?`, `GUID` y `GUID?`.

LINQ to XML proporciona los mismos operadores de conversión para los objetos XAttribute.

Ejemplo de la propiedad Value

Puede usar la propiedad Value para recuperar el contenido de un elemento:

```
XElement e = new XElement("StringElement", "abcde");
Console.WriteLine(e);
Console.WriteLine("Value of e:" + e.Value);
```

Este ejemplo produce el siguiente resultado:

```
<StringElement>abcde</StringElement>
Value of e:abcde
```

Ejemplo de elemento que podría no existir

A veces, intenta recuperar el valor de un elemento aunque no esté seguro de que exista. En este caso, al asignar el elemento convertido a un tipo de referencia que acepta valores NULL o a un tipo de valor que acepta valores NULL, si el elemento no existe, la variable asignada se establece en `null`. El código siguiente demuestra que cuando el elemento podría existir o no, resulta más fácil usar la conversión que la propiedad Value.

```
XElement root = new XElement("Root",
    new XElement("Child1", "child 1 content"),
    new XElement("Child2", "2")
);

// The following assignments show why it is easier to use
// casting when the element might or might not exist.

string c1 = (string)root.Element("Child1");
Console.WriteLine("c1:{0}", c1 == null ? "element does not exist" : c1);

int? c2 = (int?)root.Element("Child2");
Console.WriteLine("c2:{0}", c2 == null ? "element does not exist" : c2.ToString());

string c3 = (string)root.Element("Child3");
Console.WriteLine("c3:{0}", c3 == null ? "element does not exist" : c3);

int? c4 = (int?)root.Element("Child4");
Console.WriteLine("c4:{0}", c4 == null ? "element does not exist" : c4.ToString());

Console.WriteLine();

// The following assignments show the required code when using
// the Value property when the element might or might not exist.
// Notice that this is more difficult than the casting approach.

XElement e1 = root.Element("Child1");
string v1;
if (e1 == null)
    v1 = null;
else
    v1 = e1.Value;
Console.WriteLine("v1:{0}", v1 == null ? "element does not exist" : v1);

XElement e2 = root.Element("Child2");
int? v2;
if (e2 == null)
    v2 = null;
else
    v2 = Int32.Parse(e2.Value);
Console.WriteLine("v2:{0}", v2 == null ? "element does not exist" : v2.ToString());

XElement e3 = root.Element("Child3");
string v3;
if (e3 == null)
    v3 = null;
else
    v3 = e3.Value;
Console.WriteLine("v3:{0}", v3 == null ? "element does not exist" : v3);

XElement e4 = root.Element("Child4");
int? v4;
if (e4 == null)
    v4 = null;
else
    v4 = Int32.Parse(e4.Value);
Console.WriteLine("v4:{0}", v4 == null ? "element does not exist" : v4.ToString());
```

Este código genera el siguiente resultado:

```
c1:child 1 content
c2:2
c3:element does not exist
c4:element does not exist

v1:child 1 content
v2:2
v3:element does not exist
v4:element does not exist
```

Por lo general, puede escribir un código más simple al usar la conversión para recuperar el contenido de los elementos y los atributos.

Vea también

- Ejes de LINQ to XML (C#)

Procedimiento para filtrar por nombres de elemento (LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Al llamar a uno de los métodos que devuelven IEnumerable<T> de XElement, puede filtrar por nombre de elemento.

Ejemplo

Este ejemplo recupera una colección de descendientes que se filtra para incluir solo los descendientes con el nombre especificado.

En este ejemplo se usa el siguiente documento XML: Sample XML File: Typical Purchase Order (LINQ to XML) (Archivo XML de ejemplo: pedido de compra común [LINQ to XML]).

```
XElement po = XElement.Load("PurchaseOrder.xml");
IEnumerable<XElement> items =
    from el in po.Descendants("ProductName")
    select el;
foreach(XElement prdName in items)
    Console.WriteLine(prdName.Name + ":" + (string) prdName);
```

Este código genera el siguiente resultado:

```
ProductName:Lawnmower
ProductName:Baby Monitor
```

Los otros métodos que devuelven IEnumerable<T> de colecciones XElement siguen el mismo patrón. Sus firmas son similares a Elements y Descendants. A continuación, se incluye una lista completa de los métodos que tienen firmas similares:

- Ancestors
- Descendants
- Elements
- ElementsAfterSelf
- ElementsBeforeSelf
- AncestorsAndSelf
- DescendantsAndSelf

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Pedido de compra común en un espacio de nombres.

```
XNamespace aw = "http://www.adventure-works.com";
XElement po = XElement.Load("PurchaseOrderInNamespace.xml");
IEnumerable<XElement> items =
    from el in po.Descendants(aw + "ProductName")
    select el;
foreach (XElement prdName in items)
    Console.WriteLine(prdName.Name + ":" + (string)prdName);
```

Este código genera el siguiente resultado:

```
{http://www.adventure-works.com}ProductName:Lawnmower
{http://www.adventure-works.com}ProductName:Baby Monitor
```

Vea también

- LINQ to XML Axes (C#) (Ejes de LINQ to XML [C#])

Procedimiento para encadenar llamadas de métodos de eje (LINQ to XML) (C#)

18/03/2020 • 3 minutes to read • Edit Online

Un patrón común que puede utilizar en el código consiste en llamar a un método Axis y, después, llamar a uno de los métodos de extensión Axes.

Hay dos métodos Axes con el nombre `Elements` que devuelven una colección de elementos: el método XContainer.Elements y el método Extensions.Elements. Puede combinar estos dos ejes para encontrar todos los elementos de un nombre especificado en una profundidad determinada del árbol.

Ejemplo

En este ejemplo se usa XContainer.Elements y Extensions.Elements para buscar todos los elementos `Name` en todos los elementos `Address` de todos los elementos `PurchaseOrder`.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: varios pedidos de compra (LINQ to XML).

```
XElement purchaseOrders = XElement.Load("PurchaseOrders.xml");
IEnumerable<XElement> names =
    from el in purchaseOrders
        .Elements("PurchaseOrder")
        .Elements("Address")
        .Elements("Name")
    select el;
foreach (XElement e in names)
    Console.WriteLine(e);
```

Este ejemplo produce el siguiente resultado:

```
<Name>Ellen Adams</Name>
<Name>Tai Yee</Name>
<Name>Cristian Osorio</Name>
<Name>Cristian Osorio</Name>
<Name>Jessica Arnold</Name>
<Name>Jessica Arnold</Name>
```

Esto funciona porque una de las implementaciones del eje `Elements` es un método de extensión en IEnumerable<T> de XContainer. XElement se deriva de XContainer, de modo que puede llamar al método Extensions.Elements según los resultados de una llamada al método XContainer.Elements.

Ejemplo

A veces, desea recuperar todos los elementos de una profundidad determinada cuando es posible que intervengan (o no) antecesores. Por ejemplo, en el siguiente documento, podría recuperar todos los elementos `ConfigParameter` que son secundarios del elemento `Customer`, pero no el elemento `ConfigParameter` que es un secundario del elemento `Root`.

```
<Root>
  <ConfigParameter>RootConfigParameter</ConfigParameter>
  <Customer>
    <Name>Frank</Name>
    <Config>
      <ConfigParameter>FirstConfigParameter</ConfigParameter>
    </Config>
  </Customer>
  <Customer>
    <Name>Bob</Name>
    <!--This customer doesn't have a Config element-->
  </Customer>
  <Customer>
    <Name>Bill</Name>
    <Config>
      <ConfigParameter>SecondConfigParameter</ConfigParameter>
    </Config>
  </Customer>
</Root>
```

Para ello, puede usar el eje Extensions.Elements de la siguiente manera:

```
XElement root = XElement.Load("Irregular.xml");
IEnumerable<XElement> configParameters =
    root.Elements("Customer").Elements("Config").
    Elements("ConfigParameter");
foreach (XElement cp in configParameters)
    Console.WriteLine(cp);
```

Este ejemplo produce el siguiente resultado:

```
<ConfigParameter>FirstConfigParameter</ConfigParameter>
<ConfigParameter>SecondConfigParameter</ConfigParameter>
```

Ejemplo

El siguiente ejemplo muestra la misma técnica para XML que se encuentre en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Varios pedidos de compra en un espacio de nombres.

```
XNamespace aw = "http://www.adventure-works.com";
XElement purchaseOrders = XElement.Load("PurchaseOrdersInNamespace.xml");
IEnumerable<XElement> names =
    from el in purchaseOrders
        .Elements(aw + "PurchaseOrder")
        .Elements(aw + "Address")
        .Elements(aw + "Name")
    select el;
foreach (XElement e in names)
    Console.WriteLine(e);
```

Este ejemplo produce el siguiente resultado:

```
<aw:Name xmlns:aw="http://www.adventure-works.com">Ellen Adams</aw:Name>
<aw:Name xmlns:aw="http://www.adventure-works.com">Tai Yee</aw:Name>
<aw:Name xmlns:aw="http://www.adventure-works.com">Cristian Osorio</aw:Name>
<aw:Name xmlns:aw="http://www.adventure-works.com">Cristian Osorio</aw:Name>
<aw:Name xmlns:aw="http://www.adventure-works.com">Jessica Arnold</aw:Name>
<aw:Name xmlns:aw="http://www.adventure-works.com">Jessica Arnold</aw:Name>
```

Vea también

- LINQ to XML Axes (C#) (Ejes de LINQ to XML [C#])

Procedimiento para recuperar un único elemento secundario (LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se explica cómo recuperar un único elemento secundario, dado el nombre del elemento secundario. Si conoce el nombre del elemento secundario y que solo hay un elemento que tiene ese nombre, puede resultar cómodo recuperar solamente un elemento, en lugar de una recopilación.

El método Element devuelve el primer XElement secundario con el XName especificado.

Si desea recuperar un único elemento secundario de Visual Basic, un enfoque común consiste en utilizar la propiedad XML y después recuperar el primer elemento mediante una notación de indizador de matriz.

Ejemplo

En el siguiente ejemplo se muestra el uso del método Element. Este ejemplo toma un árbol XML con el nombre `po` y busca el primer elemento con el nombre `Comment`.

El ejemplo de Visual Basic muestra el uso de la notación del indizador de matriz para recuperar un elemento único.

En este ejemplo se usa el siguiente documento XML: Sample XML File: Typical Purchase Order (LINQ to XML) (Archivo XML de ejemplo: pedido de compra común [LINQ to XML]).

```
XElement po = XElement.Load("PurchaseOrder.xml");
XElement e = po.Element("DeliveryNotes");
Console.WriteLine(e);
```

Este ejemplo produce el siguiente resultado:

```
<DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
```

Ejemplo

El siguiente ejemplo muestra el mismo código sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Pedido de compra común en un espacio de nombres.

```
XElement po = XElement.Load("PurchaseOrderInNamespace.xml");
XNamespace aw = "http://www.adventure-works.com";
XElement e = po.Element(aw + "DeliveryNotes");
Console.WriteLine(e);
```

Este ejemplo produce el siguiente resultado:

```
<aw:DeliveryNotes xmlns:aw="http://www.adventure-works.com">Please leave packages in shed by driveway.
</aw:DeliveryNotes>
```

Vea también

- LINQ to XML Axes (C#) (Ejes de LINQ to XML [C#])

Procedimiento para recuperar una colección de atributos (LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se presenta el método Attributes. Este método recupera los atributos de un elemento.

Ejemplo

En el ejemplo siguiente se muestra cómo procesar una iteración en la colección de atributos de un elemento.

```
XElement val = new XElement("Value",
    new XAttribute("ID", "1243"),
    new XAttribute("Type", "int"),
    new XAttribute("ConvertableTo", "double"),
    "100");
IEnumerable<XAttribute> listOfAttributes =
    from att in val.Attributes()
    select att;
foreach (XAttribute a in listOfAttributes)
    Console.WriteLine(a);
```

Este código genera el siguiente resultado:

```
ID="1243"
Type="int"
ConvertableTo="double"
```

Vea también

- LINQ to XML Axes (C#) (Ejes de LINQ to XML [C#])

Procedimiento para recuperar un único atributo (LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema explica cómo recuperar un atributo concreto de un elemento, proporcionando el nombre del atributo. Esto puede resultar útil para escribir expresiones de consulta mediante las cuales encontrar un elemento que contiene un atributo en particular.

El método Attribute de la clase XElement devuelve el XAttribute con el nombre especificado.

Ejemplo

El siguiente ejemplo utiliza el método Attribute.

```
XElement cust = new XElement("PhoneNumbers",
    new XElement("Phone",
        new XAttribute("type", "home"),
        "555-555-5555"),
    new XElement("Phone",
        new XAttribute("type", "work"),
        "555-555-6666")
);
IEnumerable<XElement> elList =
    from el in cust.Descendants("Phone")
    select el;
foreach (XElement el in elList)
    Console.WriteLine((string)el.Attribute("type"));
```

Este ejemplo encuentra todos los descendientes en el árbol cuyo nombre es `Phone` y, a continuación, encuentra aquel atributo cuyo nombre es `type`.

Este código genera el siguiente resultado:

```
home
work
```

Ejemplo

Si desea recuperar el valor del atributo, puede convertirlo, de la misma forma que lo haría con los objetos XElement. En el siguiente ejemplo se muestra cómo hacerlo.

```
XElement cust = new XElement("PhoneNumbers",
    new XElement("Phone",
        new XAttribute("type", "home"),
        "555-555-5555"),
    new XElement("Phone",
        new XAttribute("type", "work"),
        "555-555-6666")
);
IEnumerable<XElement> elList =
    from el in cust.Descendants("Phone")
    select el;
foreach (XElement el in elList)
    Console.WriteLine((string)el.Attribute("type"));
```

Este código genera el siguiente resultado:

```
home
work
```

LINQ to XML proporciona operadores de conversión explícita para la clase XAttribute a `string`, `bool`, `bool?`, `int`, `int?`, `uint`, `uint?`, `long`, `long?`, `ulong`, `ulong?`, `float`, `float?`, `double`, `double?`, `decimal`, `decimal?`, `DateTime`, `DateTime?`, `TimeSpan`, `TimeSpan?`, `GUID` y `GUID?`.

Ejemplo

El siguiente ejemplo muestra el mismo código para un atributo que se encuentre en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

```
XNamespace aw = "http://www.adventure-works.com";
XElement cust = new XElement(aw + "PhoneNumbers",
    new XElement(aw + "Phone",
        new XAttribute(aw + "type", "home"),
        "555-555-5555"),
    new XElement(aw + "Phone",
        new XAttribute(aw + "type", "work"),
        "555-555-6666")
);
IEnumerable<XElement> elList =
    from el in cust.Descendants(aw + "Phone")
    select el;
foreach (XElement el in elList)
    Console.WriteLine((string)el.Attribute(aw + "type"));
```

Este código genera el siguiente resultado:

```
home
work
```

Vea también

- LINQ to XML Axes (C#) (Ejes de LINQ to XML [C#])

Procedimiento para recuperar el valor de un atributo (LINQ to XML) (C#)

02/04/2020 • 2 minutes to read • Edit Online

Este tema muestra cómo obtener el valor de los atributos. Existen dos formas principales: puede convertir un elemento XAttribute al tipo deseado; el operador de conversión explícita convierte el contenido del elemento o del atributo al tipo especificado. Como alternativa, puede usar la propiedad Value. Sin embargo, la conversión suele ser el mejor método. Si convierte el atributo a un tipo que admite valores NULL, resulta más fácil escribir el código al recuperar el valor de un atributo que podría existir o no. Para ver ejemplos de esta técnica, consulte Procedimiento para recuperar el valor de un elemento (LINQ to XML) (C#).

Ejemplo

Para recuperar el valor de un atributo, basta con convertir el objeto XAttribute al tipo deseado.

```
XElement root = new XElement("Root",
                    new XAttribute("Attr", "abcde")
                );
Console.WriteLine(root);
string str = (string)root.Attribute("Attr");
Console.WriteLine(str);
```

Este ejemplo produce el siguiente resultado:

```
<Root Attr="abcde" />
abcde
```

Ejemplo

El siguiente ejemplo muestra cómo recuperar el valor de un atributo, en el que el atributo se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

```
XNamespace aw = "http://www.adventure-works.com";
XElement root = new XElement(aw + "Root",
                    new XAttribute(aw + "Attr", "abcde")
                );
string str = (string)root.Attribute(aw + "Attr");
Console.WriteLine(str);
```

Este ejemplo produce el siguiente resultado:

```
abcde
```

Vea también

- Ejes de LINQ to XML (C#)

Procedimiento para recuperar el valor superficial de un elemento (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema muestra cómo obtener el valor superficial de un elemento. El valor superficial es el valor del elemento específico solamente, en oposición al valor profundo, que incluye los valores de todos los elementos descendientes concatenados en una sola cadena.

Cuando se recupera el valor de un elemento utilizando la conversión o la propiedad XElement.Value, se recupera el valor profundo. Para recuperar el valor superficial, se puede usar el método de extensión `ShallowValue`, como se muestra en el ejemplo siguiente. La recuperación del valor superficial es útil cuando se desea seleccionar elementos en función de su contenido.

En el ejemplo siguiente se declara un método de extensión que recupera el valor superficial de un elemento. A continuación, se utiliza el método de extensión en una consulta para enumerar todos los elementos que contienen un valor calculado.

Ejemplo

El siguiente archivo de texto, Report.xml, es el origen de este ejemplo.

```
<?xml version="1.0" encoding="utf-8" ?>
<Report>
  <Section>
    <Heading>
      <Column Name="CustomerId">=Customer.CustomerId.Heading</Column>
      <Column Name="Name">=Customer.Name.Heading</Column>
    </Heading>
    <Detail>
      <Column Name="CustomerId">=Customer.CustomerId</Column>
      <Column Name="Name">=Customer.Name</Column>
    </Detail>
  </Section>
</Report>
```

```
public static class MyExtensions
{
    public static string ShallowValue(this XElement xe)
    {
        return xe
               .Nodes()
               .OfType<XText>()
               .Aggregate(new StringBuilder(),
                          (s, c) => s.Append(c),
                          s => s.ToString());
    }
}

class Program
{
    static void Main(string[] args)
    {
        XElement root = XElement.Load("Report.xml");

        IEnumerable<XElement> query = from el in root.Descendants()
                                      where el.ShallowValue().StartsWith("=")
                                      select el;

        foreach (var q in query)
        {
            Console.WriteLine("{0}{1}{2}",
                q.Name.ToString().PadRight(8),
                q.Attribute("Name").ToString().PadRight(20),
                q.ShallowValue());
        }
    }
}
```

Este ejemplo produce el siguiente resultado:

```
Column  Name="CustomerId"   =Customer.CustomerId.Heading
Column  Name="Name"         =Customer.Name.Heading
Column  Name="CustomerId"   =Customer.CustomerId
Column  Name="Name"         =Customer.Name
```

Vea también

- LINQ to XML Axes (C#) (Ejes de LINQ to XML [C#])

Procedimiento para buscar un elemento con un atributo específico (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo buscar un elemento que tiene un atributo con un valor específico.

Ejemplo

El ejemplo muestra cómo buscar el elemento `Address` que tiene un atributo `Type` con un valor de "Billing".

En este ejemplo se usa el siguiente documento XML: Sample XML File: Typical Purchase Order (LINQ to XML) (Archivo XML de ejemplo: pedido de compra común [LINQ to XML]).

```
XElement root = XElement.Load("PurchaseOrder.xml");
IEnumerable<XElement> address =
    from el in root.Elements("Address")
    where (string)el.Attribute("Type") == "Billing"
    select el;
foreach (XElement el in address)
    Console.WriteLine(el);
```

Este código genera el siguiente resultado:

```
<Address Type="Billing">
  <Name>Tai Yee</Name>
  <Street>8 Oak Avenue</Street>
  <City>Old Town</City>
  <State>PA</State>
  <Zip>95819</Zip>
  <Country>USA</Country>
</Address>
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Pedido de compra común en un espacio de nombres.

```
XElement root = XElement.Load("PurchaseOrderInNamespace.xml");
XNamespace aw = "http://www.adventure-works.com";
IEnumerable<XElement> address =
    from el in root.Elements(aw + "Address")
    where (string)el.Attribute(aw + "Type") == "Billing"
    select el;
foreach (XElement el in address)
    Console.WriteLine(el);
```

Este código genera el siguiente resultado:

```
<aw:Address aw:Type="Billing" mlns:aw="http://www.adventure-works.com">
  <aw:Name>Tai Yee</aw:Name>
  <aw:Street>8 Oak Avenue</aw:Street>
  <aw:City>Old Town</aw:City>
  <aw:State>PA</aw:State>
  <aw:Zip>95819</aw:Zip>
  <aw:Country>USA</aw:Country>
</aw:Address>
```

Vea también

- Attribute
- Elements
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Projection Operations (C#) (Operaciones de proyección [C#])

Procedimiento para buscar un elemento con un elemento secundario específico (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema muestra cómo encontrar un elemento en particular que tenga un elemento secundario con un valor específico.

Ejemplo

El ejemplo busca el elemento `Test` que tenga un elemento secundario `CommandLine` con el valor de "Examp2.EXE".

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Configuración de prueba (LINQ to XML).

```
XElement root = XElement.Load("TestConfig.xml");
IEnumerable<XElement> tests =
    from el in root.Elements("Test")
    where (string)el.Element("CommandLine") == "Examp2.EXE"
    select el;
foreach (XElement el in tests)
    Console.WriteLine((string)el.Attribute("TestId"));
```

Este código genera el siguiente resultado:

```
0002
0006
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Configuración de prueba en un espacio de nombres.

```
XElement root = XElement.Load("TestConfigInNamespace.xml");
XNamespace ad = "http://www.adatum.com";
IEnumerable<XElement> tests =
    from el in root.Elements(ad + "Test")
    where (string)el.Element(ad + "CommandLine") == "Examp2.EXE"
    select el;
foreach (XElement el in tests)
    Console.WriteLine((string)el.Attribute("TestId"));
```

Este código genera el siguiente resultado:

```
0002
0006
```

Vea también

- Attribute
- Elements
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Projection Operations (C#) (Operaciones de proyección [C#])

Diferencias entre realizar consultas de un XDocument Realizar consultas de un XElement (C#)

18/03/2020 • 3 minutes to read • Edit Online

Cuando carga un documento mediante XDocument.Load, observará que es necesario escribir las consultas de forma ligeramente diferente a como lo haría en caso de cargar el documento mediante XElement.Load.

Comparación entre XDocument.Load y XElement.Load

Cuando carga un documento XML en un XElement mediante XElement.Load, el XElement situado en la raíz del árbol XML contiene al elemento raíz del documento que se ha cargado. Sin embargo, cuando carga el mismo documento XML en un XDocument mediante XDocument.Load, la raíz del árbol es un nodo XDocument el elemento raíz del documento cargado es el nodo secundario permitido XElement de XDocument. Los ejes de LINQ to XML funcionan en relación al nodo raíz.

Este primer ejemplo carga un árbol XML utilizando Load. A continuación, consulta los elementos secundarios de la raíz del árbol.

```
// Create a simple document and write it to a file
File.WriteAllText("Test.xml", @"<Root>
    <Child1>1</Child1>
    <Child2>2</Child2>
    <Child3>3</Child3>
</Root>");

Console.WriteLine("Querying tree loaded with XElement.Load");
Console.WriteLine("----");
XElement doc = XElement.Load("Test.xml");
IEnumerable<XElement> childList =
    from el in doc.Elements()
    select el;
foreach (XElement e in childList)
    Console.WriteLine(e);
```

Como cabe esperar, este ejemplo genera el siguiente resultado:

```
Querying tree loaded with XElement.Load
----
<Child1>1</Child1>
<Child2>2</Child2>
<Child3>3</Child3>
```

El siguiente ejemplo es igual al anterior, con la diferencia de que el árbol XML se carga en un XDocument en vez de un XElement.

```
// Create a simple document and write it to a file
File.WriteAllText("Test.xml", @"<Root>
    <Child1>1</Child1>
    <Child2>2</Child2>
    <Child3>3</Child3>
</Root>");

Console.WriteLine("Querying tree loaded with XDocument.Load");
Console.WriteLine("----");
XDocument doc = XDocument.Load("Test.xml");
IEnumerable<XElement> childList =
    from el in doc.Elements()
    select el;
foreach (XElement e in childList)
    Console.WriteLine(e);
```

Este ejemplo produce el siguiente resultado:

```
Querying tree loaded with XDocument.Load
----
<Root>
  <Child1>1</Child1>
  <Child2>2</Child2>
  <Child3>3</Child3>
</Root>
```

Observe que la misma consulta devolvió el nodo `Root` en vez de los tres nodos secundarios.

Una forma de afrontar este asunto es utilizar la propiedad Root antes de obtener acceso a los métodos de los ejes, tal y como se indica a continuación:

```
// Create a simple document and write it to a file
File.WriteAllText("Test.xml", @"<Root>
    <Child1>1</Child1>
    <Child2>2</Child2>
    <Child3>3</Child3>
</Root>");

Console.WriteLine("Querying tree loaded with XDocument.Load");
Console.WriteLine("----");
XDocument doc = XDocument.Load("Test.xml");
IEnumerable<XElement> childList =
    from el in doc.Root.Elements()
    select el;
foreach (XElement e in childList)
    Console.WriteLine(e);
```

Ahora, esta consulta genera el mismo resultado que la consulta del árbol cuya raíz comienza en XElement. El ejemplo genera el siguiente resultado:

```
Querying tree loaded with XDocument.Load
----
<Child1>1</Child1>
<Child2>2</Child2>
<Child3>3</Child3>
```

Procedimiento para buscar descendientes con un nombre de elemento específico (C#)

18/03/2020 • 2 minutes to read • Edit Online

A veces, desea encontrar todos los descendientes con un nombre determinado. Podría escribir código para procesar una iteración en todos los descendientes, pero es más fácil usar el eje Descendants.

Ejemplo

El ejemplo siguiente muestra cómo encontrar descendientes según el nombre de elemento.

```
XElement root = XElement.Parse(@"<root>
  <para>
    <r>
      <t>Some text </t>
    </r>
    <n>
      <r>
        <t>that is broken up into </t>
      </r>
    </n>
    <n>
      <r>
        <t>multiple segments.</t>
      </r>
    </n>
  </para>
</root>");
IEnumerable<string> textSegs =
    from seg in root.Descendants("t")
    select (string)seg;

string str = textSegs.Aggregate(new StringBuilder(),
    (sb, i) => sb.Append(i),
    sp => sp.ToString()
);

Console.WriteLine(str);
```

Este código genera el siguiente resultado:

```
Some text that is broken up into multiple segments.
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

```
XElement root = XElement.Parse(@"<root xmlns='http://www.adatum.com'>
  <para>
    <r>
      <t>Some text </t>
    </r>
    <n>
      <r>
        <t>that is broken up into </t>
      </r>
    </n>
    <n>
      <r>
        <t>multiple segments.</t>
      </r>
    </n>
  </para>
</root>");
XNamespace ad = "http://www.adatum.com";
IEnumerable<string> textSegs =
    from seg in root.Descendants(ad + "t")
    select (string)seg;

string str = textSegs.Aggregate(new StringBuilder(),
    (sb, i) => sb.Append(i),
    sp => sp.ToString()
);

Console.WriteLine(str);
```

Este código genera el siguiente resultado:

```
Some text that is broken up into multiple segments.
```

Vea también

- Descendants

Procedimiento para buscar un único descendiente con el método Descendants (C#)

18/03/2020 • 2 minutes to read • Edit Online

Puede utilizar el método de eje Descendants para escribir rápidamente código para buscar un solo elemento cuyo nombre es único. Esta técnica es especialmente útil si desea buscar un descendiente particular con un nombre específico. Puede escribir el código para desplazarse al elemento deseado, pero a menudo resulta más rápido escribir el código usando el eje Descendants.

Ejemplo

Este ejemplo utiliza el operador de consulta estándar First.

```
XElement root = XElement.Parse(@"<Root>
  <Child1>
    <GrandChild1>GC1 Value</GrandChild1>
  </Child1>
  <Child2>
    <GrandChild2>GC2 Value</GrandChild2>
  </Child2>
  <Child3>
    <GrandChild3>GC3 Value</GrandChild3>
  </Child3>
  <Child4>
    <GrandChild4>GC4 Value</GrandChild4>
  </Child4>
</Root>");
string grandChild3 = (string)
    (from el in root.Descendants("GrandChild3")
    select el).First();
Console.WriteLine(grandChild3);
```

Este código genera el siguiente resultado:

```
GC3 Value
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

```
XElement root = XElement.Parse(@"<aw:Root xmlns:aw='http://www.adventure-works.com'>
  <aw:Child1>
    <aw:GrandChild1>GC1 Value</aw:GrandChild1>
  </aw:Child1>
  <aw:Child2>
    <aw:GrandChild2>GC2 Value</aw:GrandChild2>
  </aw:Child2>
  <aw:Child3>
    <aw:GrandChild3>GC3 Value</aw:GrandChild3>
  </aw:Child3>
  <aw:Child4>
    <aw:GrandChild4>GC4 Value</aw:GrandChild4>
  </aw:Child4>
</aw:Root>");
XNamespace aw = "http://www.adventure-works.com";
string grandChild3 = (string)
    (from el in root.Descendants(aw + "GrandChild3")
     select el).First();
Console.WriteLine(grandChild3);
```

Este código genera el siguiente resultado:

```
GC3 Value
```

Procedimiento para escribir consultas con filtrado complejo (C#)

18/03/2020 • 2 minutes to read • Edit Online

Es posible que desee escribir consultas LINQ to XML con filtros complejos. Por ejemplo, quizás debe buscar todos los elementos que tienen un elemento secundario con un valor y un nombre específicos. En este tema se proporciona un ejemplo de escritura de una consulta con un filtrado complejo.

Ejemplo

En este ejemplo se muestra cómo buscar todos los elementos `PurchaseOrder` que tienen un elemento `Address` secundario que tiene un atributo `Type` igual a "Shipping" (envío) y un elemento `State` secundario igual a "NY". Utiliza una consulta anidada en la cláusula `Where` y el operador `Any` devuelve `true` si la colección tiene elementos en ella. Para obtener información sobre cómo usar la sintaxis de consultas basadas en métodos, vea Query Syntax and Method Syntax in LINQ (Sintaxis de consultas y sintaxis de métodos en LINQ).

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: varios pedidos de compra (LINQ to XML).

Para obtener más información sobre el operador `Any`, vea Quantifier Operations (C#) (Operaciones cuantificadoras (C#)).

```
XElement root = XElement.Load("PurchaseOrders.xml");
IEnumerable<XElement> purchaseOrders =
    from el in root.Elements("PurchaseOrder")
    where
        (from add in el.Elements("Address")
        where
            (string)add.Attribute("Type") == "Shipping" &&
            (string)add.Element("State") == "NY"
        select add)
        .Any()
    select el;
foreach (XElement el in purchaseOrders)
    Console.WriteLine((string)el.Attribute("PurchaseOrderNumber"));
```

Este código genera el siguiente resultado:

```
99505
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Varios pedidos de compra en un espacio de nombres.

```
XElement root = XElement.Load("PurchaseOrdersInNamespace.xml");
XNamespace aw = "http://www.adventure-works.com";
IEnumerable<XElement> purchaseOrders =
    from el in root.Elements(aw + "PurchaseOrder")
    where
        (from add in el.Elements(aw + "Address")
         where
             (string)add.Attribute(aw + "Type") == "Shipping" &&
             (string)add.Element(aw + "State") == "NY"
         select add)
        .Any()
    select el;
foreach (XElement el in purchaseOrders)
    Console.WriteLine((string)el.Attribute(aw + "PurchaseOrderNumber"));
```

Este código genera el siguiente resultado:

```
99505
```

Vea también

- Attribute
- Elements
- Projection Operations (C#) (Operaciones de proyección [C#])
- Quantifier Operations (C#) (Operaciones cuantificadoras (C#))

Procedimiento para filtrar por un elemento opcional (C#)

18/03/2020 • 2 minutes to read • Edit Online

En ocasiones, deseará filtrar por un elemento dado a pesar de que no está seguro de si existe o no en el documento XML. La consulta debería ejecutarse de forma que si el elemento en particular no tiene ningún elemento secundario, no se produzca una excepción de referencia nula al filtrar por él. En el ejemplo siguiente, el elemento `Child5` no tiene ningún elemento secundario `Type`, pero aún así, la consulta se ejecuta correctamente.

Ejemplo

Este ejemplo utiliza el método de extensión Elements.

```
XElement root = XElement.Parse(@"<Root>
  <Child1>
    <Text>Child One Text</Text>
    <Type Value=""Yes""/>
  </Child1>
  <Child2>
    <Text>Child Two Text</Text>
    <Type Value=""Yes""/>
  </Child2>
  <Child3>
    <Text>Child Three Text</Text>
    <Type Value=""No""/>
  </Child3>
  <Child4>
    <Text>Child Four Text</Text>
    <Type Value=""Yes""/>
  </Child4>
  <Child5>
    <Text>Child Five Text</Text>
  </Child5>
</Root>");
var cList =
    from typeElement in root.Elements().Elements("Type")
    where (string)typeElement.Attribute("Value") == "Yes"
    select (string)typeElement.Parent.Element("Text");
foreach(string str in cList)
    Console.WriteLine(str);
```

Este código genera el siguiente resultado:

```
Child One Text
Child Two Text
Child Four Text
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

```
XElement root = XElement.Parse(@"<Root xmlns='http://www.adatum.com'>
  <Child1>
    <Text>Child One Text</Text>
    <Type Value=""Yes""/>
  </Child1>
  <Child2>
    <Text>Child Two Text</Text>
    <Type Value=""Yes""/>
  </Child2>
  <Child3>
    <Text>Child Three Text</Text>
    <Type Value=""No""/>
  </Child3>
  <Child4>
    <Text>Child Four Text</Text>
    <Type Value=""Yes""/>
  </Child4>
  <Child5>
    <Text>Child Five Text</Text>
  </Child5>
</Root>");
XNamespace ad = "http://www.adatum.com";
var cList =
    from typeElement in root.Elements().Elements(ad + "Type")
    where (string)typeElement.Attribute("Value") == "Yes"
    select (string)typeElement.Parent.Element(ad + "Text");
foreach (string str in cList)
    Console.WriteLine(str);
```

Este código genera el siguiente resultado:

```
Child One Text
Child Two Text
Child Four Text
```

Vea también

- XElement.Attribute
- XContainer.Elements
- Extensions.Elements
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- Projection Operations (C#) (Operaciones de proyección [C#])

Procedimiento para buscar todos los nodos de un espacio de nombres (C#)

18/03/2020 • 2 minutes to read • Edit Online

Puede filtrar en el espacio de nombres de cada elemento o atributo para buscar todos los nodos de ese espacio de nombres particular.

Ejemplo

En el ejemplo siguiente se crea un árbol XML con dos espacios de nombres. A continuación, recorre en iteración los árboles y muestra los nombres de todos los elementos y atributos de esos espacios de nombres.

```
string markup = @"<aw:Root xmlns:aw='http://www.adventure-works.com' xmlns:fc='www.fourthcoffee.com'>
  <fc:Child1>abc</fc:Child1>
  <fc:Child2>def</fc:Child2>
  <aw:Child3>ghi</aw:Child3>
  <fc:Child4>
    <fc:GrandChild1>jkl</fc:GrandChild1>
    <aw:GrandChild2>mno</aw:GrandChild2>
  </fc:Child4>
</aw:Root>";
XElement xmlTree = XElement.Parse(markup);
Console.WriteLine("Nodes in the http://www.adventure-works.com namespace");
IEnumerable<XElement> awElements =
    from el in xmlTree.Descendants()
    where el.Name.Namespace == "http://www.adventure-works.com"
    select el;
foreach (XElement el in awElements)
    Console.WriteLine(el.Name.ToString());
```

Este código genera el siguiente resultado:

```
Nodes in the http://www.adventure-works.com namespace
{http://www.adventure-works.com}Child3
{http://www.adventure-works.com}GrandChild2
```

Ejemplo

El archivo XML al que tiene acceso la siguiente consulta contiene pedidos de compra en dos espacios de nombres diferentes. La consulta crea un nuevo árbol con solo los elementos de uno de los espacios de nombres.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Pedidos de compra consolidados.

```
XDocument cpo = XDocument.Load("ConsolidatedPurchaseOrders.xml");
XNamespace aw = "http://www.adventure-works.com";
XElement newTree = new XElement("Root",
    from el in cpo.Root.Elements()
    where el.Name.Namespace == aw
    select el
);
Console.WriteLine(newTree);
```

Este código genera el siguiente resultado:

```
<Root>
  <aw:PurchaseOrder PONumber="11223" Date="2000-01-15" xmlns:aw="http://www.adventure-works.com">
    <aw:ShippingAddress>
      <aw:Name>Chris Preston</aw:Name>
      <aw:Street>123 Main St.</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98113</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:ShippingAddress>
    <aw:BillingAddress>
      <aw:Name>Chris Preston</aw:Name>
      <aw:Street>123 Main St.</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98113</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:BillingAddress>
    <aw:DeliveryInstructions>Ship only complete order.</aw:DeliveryInstructions>
    <aw:Item PartNum="LIT-01">
      <aw:ProductID>Litware Networking Card</aw:ProductID>
      <aw:Qty>1</aw:Qty>
      <aw:Price>20.99</aw:Price>
    </aw:Item>
    <aw:Item PartNum="LIT-25">
      <aw:ProductID>Litware 17in LCD Monitor</aw:ProductID>
      <aw:Qty>1</aw:Qty>
      <aw:Price>199.99</aw:Price>
    </aw:Item>
  </aw:PurchaseOrder>
</Root>
```

Procedimiento para ordenar elementos (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este ejemplo muestra cómo escribir una consulta que ordene sus resultados.

Ejemplo

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Datos numéricos (LINQ to XML).

```
XElement root = XElement.Load("Data.xml");
IEnumerable<decimal> prices =
    from el in root.Elements("Data")
    let price = (decimal)el.Element("Price")
    orderby price
    select price;
foreach (decimal el in prices)
    Console.WriteLine(el);
```

Este código genera el siguiente resultado:

```
0.99
4.95
6.99
24.50
29.00
66.00
89.99
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Datos numéricos en un espacio de nombres.

```
XElement root = XElement.Load("DataInNamespace.xml");
XNamespace aw = "http://www.adatum.com";
IEnumerable<decimal> prices =
    from el in root.Elements(aw + "Data")
    let price = (decimal)el.Element(aw + "Price")
    orderby price
    select price;
foreach (decimal el in prices)
    Console.WriteLine(el);
```

Este código genera el siguiente resultado:

```
0.99
4.95
6.99
24.50
29.00
66.00
89.99
```

Vea también

- Ordenación de datos [C#]

Procedimiento para ordenar elementos por varias claves (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se describe cómo ordenar en varias claves.

Ejemplo

En este ejemplo los resultados se ofrecen primero por código postal de envío y después por fecha de ordenación.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

```
XElement co = XElement.Load("CustomersOrders.xml");
var sortedElements =
    from c in co.Element("Orders").Elements("Order")
    orderby (string)c.Element("ShipInfo").Element("ShipPostalCode"),
            (DateTime)c.Element("OrderDate")
    select new {
        CustomerID = (string)c.Element("CustomerID"),
        EmployeeID = (string)c.Element("EmployeeID"),
        ShipPostalCode = (string)c.Element("ShipInfo").Element("ShipPostalCode"),
        OrderDate = (DateTime)c.Element("OrderDate")
    };
foreach (var r in sortedElements)
    Console.WriteLine("CustomerID:{0} EmployeeID:{1} ShipPostalCode:{2} OrderDate:{3:d}",
        r.CustomerID, r.EmployeeID, r.ShipPostalCode, r.OrderDate);
```

Este código genera el siguiente resultado:

```
CustomerID:LETSS EmployeeID:1 ShipPostalCode:94117 OrderDate:6/25/1997
CustomerID:LETSS EmployeeID:8 ShipPostalCode:94117 OrderDate:10/27/1997
CustomerID:LETSS EmployeeID:6 ShipPostalCode:94117 OrderDate:11/10/1997
CustomerID:LETSS EmployeeID:4 ShipPostalCode:94117 OrderDate:2/12/1998
CustomerID:GREAL EmployeeID:6 ShipPostalCode:97403 OrderDate:5/6/1997
CustomerID:GREAL EmployeeID:8 ShipPostalCode:97403 OrderDate:7/4/1997
CustomerID:GREAL EmployeeID:1 ShipPostalCode:97403 OrderDate:7/31/1997
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:7/31/1997
CustomerID:GREAL EmployeeID:6 ShipPostalCode:97403 OrderDate:9/4/1997
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:9/25/1997
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:1/6/1998
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:3/9/1998
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:4/7/1998
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:4/22/1998
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:4/30/1998
CustomerID:HUNGC EmployeeID:3 ShipPostalCode:97827 OrderDate:12/6/1996
CustomerID:HUNGC EmployeeID:1 ShipPostalCode:97827 OrderDate:12/25/1996
CustomerID:HUNGC EmployeeID:3 ShipPostalCode:97827 OrderDate:1/15/1997
CustomerID:HUNGC EmployeeID:4 ShipPostalCode:97827 OrderDate:7/16/1997
CustomerID:HUNGC EmployeeID:8 ShipPostalCode:97827 OrderDate:9/8/1997
CustomerID:LAZYK EmployeeID:1 ShipPostalCode:99362 OrderDate:3/21/1997
CustomerID:LAZYK EmployeeID:8 ShipPostalCode:99362 OrderDate:5/22/1997
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para

más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos en un espacio de nombres.

```
XElement co = XElement.Load("CustomersOrdersInNamespace.xml");
XNamespace aw = "http://www.adventure-works.com";
var sortedElements =
    from c in co.Element(aw + "Orders").Elements(aw + "Order")
    orderby (string)c.Element(aw + "ShipInfo").Element(aw + "ShipPostalCode"),
            (DateTime)c.Element(aw + "OrderDate")
    select new
    {
        CustomerID = (string)c.Element(aw + "CustomerID"),
        EmployeeID = (string)c.Element(aw + "EmployeeID"),
        ShipPostalCode = (string)c.Element(aw + "ShipInfo").Element(aw + "ShipPostalCode"),
        OrderDate = (DateTime)c.Element(aw + "OrderDate")
    };
foreach (var r in sortedElements)
    Console.WriteLine("CustomerID:{0} EmployeeID:{1} ShipPostalCode:{2} OrderDate:{3:d}",
        r.CustomerID, r.EmployeeID, r.ShipPostalCode, r.OrderDate);
```

Este código genera el siguiente resultado:

```
CustomerID:LETSS EmployeeID:1 ShipPostalCode:94117 OrderDate:6/25/1997
CustomerID:LETSS EmployeeID:8 ShipPostalCode:94117 OrderDate:10/27/1997
CustomerID:LETSS EmployeeID:6 ShipPostalCode:94117 OrderDate:11/10/1997
CustomerID:LETSS EmployeeID:4 ShipPostalCode:94117 OrderDate:2/12/1998
CustomerID:GREAL EmployeeID:6 ShipPostalCode:97403 OrderDate:5/6/1997
CustomerID:GREAL EmployeeID:8 ShipPostalCode:97403 OrderDate:7/4/1997
CustomerID:GREAL EmployeeID:1 ShipPostalCode:97403 OrderDate:7/31/1997
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:7/31/1997
CustomerID:GREAL EmployeeID:6 ShipPostalCode:97403 OrderDate:9/4/1997
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:9/25/1997
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:1/6/1998
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:3/9/1998
CustomerID:GREAL EmployeeID:3 ShipPostalCode:97403 OrderDate:4/7/1998
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:4/22/1998
CustomerID:GREAL EmployeeID:4 ShipPostalCode:97403 OrderDate:4/30/1998
CustomerID:HUNGC EmployeeID:3 ShipPostalCode:97827 OrderDate:12/6/1996
CustomerID:HUNGC EmployeeID:1 ShipPostalCode:97827 OrderDate:12/25/1996
CustomerID:HUNGC EmployeeID:3 ShipPostalCode:97827 OrderDate:1/15/1997
CustomerID:HUNGC EmployeeID:4 ShipPostalCode:97827 OrderDate:7/16/1997
CustomerID:HUNGC EmployeeID:8 ShipPostalCode:97827 OrderDate:9/8/1997
CustomerID:LAZYK EmployeeID:1 ShipPostalCode:99362 OrderDate:3/21/1997
CustomerID:LAZYK EmployeeID:8 ShipPostalCode:99362 OrderDate:5/22/1997
```

Procedimiento para calcular valores intermedios (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este ejemplo muestra cómo calcular valores intermedios que se pueden usar para ordenar, filtrar y seleccionar.

Ejemplo

El siguiente ejemplo utiliza la cláusula `Let`.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Datos numéricos (LINQ to XML).

```
XElement root = XElement.Load("Data.xml");
IEnumerable<decimal> extensions =
    from el in root.Elements("Data")
    let extension = (decimal)el.Element("Quantity") * (decimal)el.Element("Price")
    where extension >= 25
    orderby extension
    select extension;
foreach (decimal ex in extensions)
    Console.WriteLine(ex);
```

Este código genera el siguiente resultado:

```
55.92
73.50
89.99
198.00
435.00
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Datos numéricos en un espacio de nombres.

```
XElement root = XElement.Load("DataInNamespace.xml");
XNamespace ad = "http://www.adatum.com";
IEnumerable<decimal> extensions =
    from el in root.Elements(ad + "Data")
    let extension = (decimal)el.Element(ad + "Quantity") * (decimal)el.Element(ad + "Price")
    where extension >= 25
    orderby extension
    select extension;
foreach (decimal ex in extensions)
    Console.WriteLine(ex);
```

Este código genera el siguiente resultado:

```
55.92
73.50
89.99
198.00
435.00
```

Procedimiento para escribir una consulta que busca elementos en función del contexto (C#)

18/03/2020 • 2 minutes to read • Edit Online

Es posible que alguna vez tenga que escribir una consulta que seleccione elementos basándose en su contexto. Quizás desea filtrar basándose en elementos de mismo nivel precedentes o siguientes. Quizás desea filtrar basándose e elementos secundarios o elementos antecesores.

Puede hacerlo escribiendo una consulta y utilizando los resultados de la consulta en la cláusula `where`. Si primero tiene que realizar una prueba con valores NULL y después probar el valor, resulta más cómodo realizar la consulta en una cláusula `let` y después utilizar los resultados en la cláusula `where`.

Ejemplo

El siguiente ejemplo selecciona todos los elementos `p` que van seguidos inmediatamente por un elemento `ul`.

```
XElement doc = XElement.Parse(@"<Root>
    <p id=""1""/>
    <ul>abc</ul>
    <Child>
        <p id=""2""/>
        <notul/>
        <p id=""3""/>
        <ul>def</ul>
        <p id=""4""/>
    </Child>
    <Child>
        <p id=""5""/>
        <notul/>
        <p id=""6""/>
        <ul>abc</ul>
        <p id=""7""/>
    </Child>
</Root>");

IEnumerable<XElement> items =
    from e in doc.Descendants("p")
    let z = e.ElementsAfterSelf().FirstOrDefault()
    where z != null && z.Name.LocalName == "ul"
    select e;

foreach (XElement e in items)
    Console.WriteLine("id = {0}", (string)e.Attribute("id"));
```

Este código genera el siguiente resultado:

```
id = 1
id = 3
id = 6
```

Ejemplo

El siguiente ejemplo muestra la misma consulta sobre un XML que se encuentra en un espacio de nombres. Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

```
XElement doc = XElement.Parse(@"<Root xmlns='http://www.adatum.com'>
    <p id=""1""/>
    <ul>abc</ul>
    <Child>
        <p id=""2""/>
        <notul/>
        <p id=""3""/>
        <ul>def</ul>
        <p id=""4""/>
    </Child>
    <Child>
        <p id=""5""/>
        <notul/>
        <p id=""6""/>
        <ul>abc</ul>
        <p id=""7""/>
    </Child>
</Root>");

XNamespace ad = "http://www.adatum.com";

IEnumerable<XElement> items =
    from e in doc.Descendants(ad + "p")
    let z = e.ElementsAfterSelf().FirstOrDefault()
    where z != null && z.Name == ad.GetName("ul")
    select e;

foreach (XElement e in items)
    Console.WriteLine("id = {0}", (string)e.Attribute("id"));
```

Este código genera el siguiente resultado:

```
id = 1
id = 3
id = 6
```

Vea también

- Parse
- Descendants
- ElementsAfterSelf
- FirstOrDefault

Procedimiento para depurar conjuntos de resultados de consulta vacíos (C#)

18/03/2020 • 2 minutes to read • Edit Online

Uno de los problemas más habituales al consultar árboles XML es que si el árbol XML tiene un espacio de nombres predeterminado, el desarrollador a veces escribe la consulta como si el código XML no estuviera en un espacio de nombres.

El primer conjunto de ejemplos de este tema muestra una forma habitual de cargar XML en un espacio de nombres predeterminado con una consulta incorrecta.

El segundo conjunto de ejemplos muestra las correcciones necesarias para que pueda consultar el código XML en un espacio de nombres.

Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

Ejemplo

Este ejemplo muestra la creación de XML en un espacio de nombres, y una consulta que devuelve un conjunto de resultados vacío.

```
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
    <Child>1</Child>
    <Child>2</Child>
    <Child>3</Child>
    <AnotherChild>4</AnotherChild>
    <AnotherChild>5</AnotherChild>
    <AnotherChild>6</AnotherChild>
</Root>");
IEnumerable<XElement> c1 =
    from el in root.Elements("Child")
    select el;
Console.WriteLine("Result set follows:");
foreach (XElement el in c1)
    Console.WriteLine((int)el);
Console.WriteLine("End of result set");
```

Este ejemplo genera el siguiente resultado:

```
Result set follows:
End of result set
```

Ejemplo

Este ejemplo muestra la creación de XML en un espacio de nombres, y una consulta que se codifica correctamente.

La solución consiste en declarar e inicializar un objeto XNamespace, y usarlo cuando se especifiquen objetos XName. En este caso, el argumento para el método Elements es un objeto XName.

```
XElement root = XElement.Parse(
@"<Root xmlns='http://www.adventure-works.com'>
    <Child>1</Child>
    <Child>2</Child>
    <Child>3</Child>
    <AnotherChild>4</AnotherChild>
    <AnotherChild>5</AnotherChild>
    <AnotherChild>6</AnotherChild>
</Root>");
XNamespace aw = "http://www.adventure-works.com";
IEnumerable<XElement> c1 =
    from el in root.Elements(aw + "Child")
    select el;
Console.WriteLine("Result set follows:");
foreach (XElement el in c1)
    Console.WriteLine((int)el);
Console.WriteLine("End of result set");
```

Este ejemplo genera el siguiente resultado:

```
Result set follows:
1
2
3
End of result set
```

Procedimiento para trabajar con diccionarios mediante LINQ to XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

A menudo resulta cómodo convertir variedades de estructuras de datos a XML y de XML a otras estructuras de datos. Este tema muestra una implementación específica de este enfoque general convirtiendo un Dictionary<TKey,TValue> a XML y de XML.

Ejemplo

En este ejemplo se usa una forma de construcción funcional en la que una consulta proyecta nuevos objetos XElement y la colección resultante se pasa como argumento al constructor del objeto raíz XElement.

```
Dictionary<string, string> dict = new Dictionary<string, string>();
dict.Add("Child1", "Value1");
dict.Add("Child2", "Value2");
dict.Add("Child3", "Value3");
dict.Add("Child4", "Value4");
XElement root = new XElement("Root",
    from keyValue in dict
    select new XElement(keyValue.Key, keyValue.Value)
);
Console.WriteLine(root);
```

Este código genera el siguiente resultado:

```
<Root>
  <Child1>Value1</Child1>
  <Child2>Value2</Child2>
  <Child3>Value3</Child3>
  <Child4>Value4</Child4>
</Root>
```

Ejemplo

El siguiente código genera un diccionario de XML.

```
XElement root = new XElement("Root",
    new XElement("Child1", "Value1"),
    new XElement("Child2", "Value2"),
    new XElement("Child3", "Value3"),
    new XElement("Child4", "Value4")
);

Dictionary<string, string> dict = new Dictionary<string, string>();
foreach (XElement el in root.Elements())
    dict.Add(el.Name.LocalName, el.Value);
foreach (string str in dict.Keys)
    Console.WriteLine("{0}:{1}", str, dict[str]);
```

Este código genera el siguiente resultado:

```
Child1:Value1
Child2:Value2
Child3:Value3
Child4:Value4
```

Procedimiento para transformar la forma de un árbol XML (C#)

18/03/2020 • 4 minutes to read • Edit Online

La *forma* de un documento XML hace referencia a sus nombres de elemento, sus nombres de atributo y las características de su jerarquía.

A veces, deberá cambiar la forma de un elemento XML. Por ejemplo, es posible que deba enviar un documento XML a otro sistema que requiere nombres de elemento y atributo diferentes. Podría revisar el documento y eliminar y cambiar el nombre de los elementos necesarios, pero el uso de la construcción funcional proporciona un código más legible y fácil de mantener. Para obtener más información sobre la construcción funcional, consulte Construcción funcional (LINQ to XML) (C#).

El primer ejemplo cambia la organización del documento XML. Mueve los elementos complejos de una ubicación del árbol a otra.

El segundo ejemplo de este tema crea un documento XML con una forma diferente a la del documento de origen. Cambia el uso de mayúsculas y minúsculas de los nombres de elemento, cambia el nombre de algunos elementos y deja algunos de los elementos del árbol de origen fuera del árbol transformado.

Ejemplo

El código siguiente cambia la forma de un archivo XML con las expresiones de consulta incrustadas.

El documento XML de origen de este ejemplo contiene un elemento `Customers` en el elemento `Root` que contiene todos los clientes. También contiene un elemento `Orders` en el elemento `Root` que contiene todos los pedidos. Este ejemplo crea un nuevo árbol XML en el que los pedidos de cada cliente se incluyen en un elemento `Orders` dentro del elemento `Customer`. El documento original también contiene un elemento `CustomerID` en el elemento `Order`; este elemento se quitará del documento con la forma cambiada.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

```
XElement co = XElement.Load("CustomersOrders.xml");
XElement newCustOrd =
    new XElement("Root",
        from cust in co.Element("Customers").Elements("Customer")
        select new XElement("Customer",
            cust.Attributes(),
            cust.Elements(),
            new XElement("Orders",
                from ord in co.Element("Orders").Elements("Order")
                where (string)ord.Element("CustomerID") == (string)cust.Attribute("CustomerID")
                select new XElement("Order",
                    ord.Attributes(),
                    ord.Element("EmployeeID"),
                    ord.Element("OrderDate"),
                    ord.Element("RequiredDate"),
                    ord.Element("ShipInfo")
                )
            )
        )
    );
Console.WriteLine(newCustOrd);
```

Este código genera el siguiente resultado:

```
<Root>
  <Customer CustomerID="GREAL">
    <CompanyName>Great Lakes Food Market</CompanyName>
    <ContactName>Howard Snyder</ContactName>
    <ContactTitle>Marketing Manager</ContactTitle>
    <Phone>(503) 555-7555</Phone>
    <FullAddress>
      <Address>2732 Baker Blvd.</Address>
      <City>Eugene</City>
      <Region>OR</Region>
      <PostalCode>97403</PostalCode>
      <Country>USA</Country>
    </FullAddress>
    <Orders />
  </Customer>
  <Customer CustomerID="HUNGC">
    <CompanyName>Hungry Coyote Import Store</CompanyName>
    <ContactName>Yoshi Latimer</ContactName>
    <ContactTitle>Sales Representative</ContactTitle>
    <Phone>(503) 555-6874</Phone>
    <Fax>(503) 555-2376</Fax>
    <FullAddress>
      <Address>City Center Plaza 516 Main St.</Address>
      <City>Elgin</City>
      <Region>OR</Region>
      <PostalCode>97827</PostalCode>
      <Country>USA</Country>
    </FullAddress>
    <Orders />
  </Customer>
  . . .
```

Ejemplo

Este ejemplo cambia el nombre de algunos elementos y convierte algunos atributos en elementos.

El código llama a `ConvertAddress`, que devuelve una lista de objetos XElement. El argumento del método es una consulta que determina el elemento complejo `Address` en el que el atributo `Type` tiene un valor `"Shipping"`.

En este ejemplo se usa el siguiente documento XML: Sample XML File: Typical Purchase Order (LINQ to XML) (Archivo XML de ejemplo: pedido de compra común [LINQ to XML]).

```
static IEnumerable<XElement> ConvertAddress(XElement add)
{
    List<XElement> fragment = new List<XElement>() {
        new XElement("NAME", (string)add.Element("Name")),
        new XElement("STREET", (string)add.Element("Street")),
        new XElement("CITY", (string)add.Element("City")),
        new XElement("ST", (string)add.Element("State")),
        new XElement("POSTALCODE", (string)add.Element("Zip")),
        new XElement("COUNTRY", (string)add.Element("Country"))
    };
    return fragment;
}

static void Main(string[] args)
{
    XElement po = XElement.Load("PurchaseOrder.xml");
    XElement newPo = new XElement("PO",
        new XElement("ID", (string)po.Attribute("PurchaseOrderNumber")),
        new XElement("DATE", (DateTime)po.Attribute("OrderDate")),
        ConvertAddress(
            (from el in po.Elements("Address")
            where (string)el.Attribute("Type") == "Shipping"
            select el)
            .First()
        )
    );
    Console.WriteLine(newPo);
}
```

Este código genera el siguiente resultado:

```
<PO>
  <ID>99503</ID>
  <DATE>1999-10-20T00:00:00</DATE>
  <NAME>Ellen Adams</NAME>
  <STREET>123 Maple Street</STREET>
  <CITY>Mill Valley</CITY>
  <ST>CA</ST>
  <POSTALCODE>10999</POSTALCODE>
  <COUNTRY>USA</COUNTRY>
</PO>
```

Procedimiento para controlar el tipo de una proyección (C#)

18/03/2020 • 2 minutes to read • Edit Online

La proyección es el proceso de tomar un conjunto de datos, filtrarlo, cambiar su forma e incluso cambiar su tipo. La mayoría de las expresiones de consulta realizan proyecciones. La mayor parte de las expresiones de consulta de esta sección se evalúan como IEnumerable<T> de XElement, aunque puede controlar el tipo de proyección para crear colecciones de otros tipos. En este tema se explica cómo hacerlo.

Ejemplo

En el ejemplo siguiente, se define un nuevo tipo, `Customer`. La expresión de consulta crea una instancia de nuevos objetos `Customer` en la cláusula `Select`. Esto hace que el tipo de la expresión de consulta sea IEnumerable<T> de `Customer`.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

```
public class Customer
{
    private string customerID;
    public string CustomerID{ get{return customerID;} set{customerID = value;}}

    private string companyName;
    public string CompanyName{ get{return companyName;} set{companyName = value;}}

    private string contactName;
    public string ContactName { get{return contactName;} set{contactName = value;}}

    public Customer(string customerID, string companyName, string contactName)
    {
        CustomerID = customerID;
        CompanyName = companyName;
        ContactName = contactName;
    }

    public override string ToString()
    {
        return $"{this.customerID}:{this.companyName}:{this.contactName}";
    }
}

class Program
{
    static void Main(string[] args)
    {
        XElement custOrd = XElement.Load("CustomersOrders.xml");
        IEnumerable<Customer> custList =
            from el in custOrd.Element("Customers").Elements("Customer")
            select new Customer(
                (string)el.Attribute("CustomerID"),
                (string)el.Element("CompanyName"),
                (string)el.Element("ContactName")
            );
        foreach (Customer cust in custList)
            Console.WriteLine(cust);
    }
}
```

Este código genera el siguiente resultado:

```
GREAL:Great Lakes Food Market:Howard Snyder
HUNGC:Hungry Coyote Import Store:Yoshi Latimer
LAZYK:Lazy K Kountry Store:John Steel
LETSS:Let's Stop N Shop:Jaime Yorres
```

Vea también

- Select

Procedimiento para proyectar un nuevo tipo (LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Otros ejemplos de esta sección han mostrado consultados que devuelven resultados como IEnumerable<T> de XElement, IEnumerable<T> de `string` y IEnumerable<T> de `int`. Se trata de tipos de resultados comunes, pero no son adecuados para cada caso. En muchos casos querrá que sus consultas devuelvan un IEnumerable<T> de algún otro tipo.

Ejemplo

En este ejemplo se muestra como crear instancias de objetos en la cláusula `select`. El código primero define una nueva clase con un constructor y después modifica la instrucción `select` para que la expresión sea una nueva instancia de la nueva clase.

En este ejemplo se usa el siguiente documento XML: Sample XML File: Typical Purchase Order (LINQ to XML) (Archivo XML de ejemplo: pedido de compra común [LINQ to XML]).

```
class NameQty
{
    public string name;
    public int qty;
    public NameQty(string n, int q)
    {
        name = n;
        qty = q;
    }
};

class Program {
    public static void Main()
    {
        XElement po = XElement.Load("PurchaseOrder.xml");

        IEnumerable<NameQty> nqList =
            from n in po.Descendants("Item")
            select new NameQty(
                    (string)n.Element("ProductName"),
                    (int)n.Element("Quantity")
                );

        foreach (NameQty n in nqList)
            Console.WriteLine(n.name + ":" + n.qty);
    }
}
```

En este ejemplo se usa el método Element que se introdujo en el tema Procedimiento para recuperar un único elemento secundario (LINQ to XML) (C#). También utiliza conversiones para recuperar los valores de los elementos devueltos por el método Element.

Este ejemplo produce el siguiente resultado:

```
Lawnmower:1
Baby Monitor:2
```

Procedimiento para proyectar un gráfico de objetos (C#)

18/03/2020 • 4 minutes to read • Edit Online

En este tema se ilustra cómo proyectar o rellenar un gráfico de objetos de XML.

Ejemplo

El siguiente código rellena un gráfico de objetos con las clases `Address`, `PurchaseOrder` y `PurchaseOrderItem` del documento XML Archivo XML de ejemplo: Pedido de compra común (LINQ to XML).

```
class Address
{
    public enum AddressUse
    {
        Shipping,
        Billing,
    }

    private AddressUse addressType;
    private string name;
    private string street;
    private string city;
    private string state;
    private string zip;
    private string country;

    public AddressUse AddressType {
        get { return addressType; } set { addressType = value; }
    }

    public string Name {
        get { return name; } set { name = value; }
    }

    public string Street {
        get { return street; } set { street = value; }
    }

    public string City {
        get { return city; } set { city = value; }
    }

    public string State {
        get { return state; } set { state = value; }
    }

    public string Zip {
        get { return zip; } set { zip = value; }
    }

    public string Country {
        get { return country; } set { country = value; }
    }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append($"Type: {(addressType == AddressUse.Shipping ? "Shipping" : "Billing")}\n");
        sb.Append($"Name: {name}\n");
```

```
        sb.Append($"Name: {name}\n");
        sb.Append($"Street: {street}\n");
        sb.Append($"City: {city}\n");
        sb.Append($"State: {state}\n");
        sb.Append($"Zip: {zip}\n");
        sb.Append($"Country: {country}\n");
        return sb.ToString();
    }
}

class PurchaseOrderItem
{
    private string partNumber;
    private string productName;
    private int quantity;
    private Decimal usPrice;
    private string comment;
    private DateTime shipDate;

    public string PartNumber {
        get { return partNumber; } set { partNumber = value; }
    }

    public string ProductName {
        get { return productName; } set { productName = value; }
    }

    public int Quantity {
        get { return quantity; } set { quantity = value; }
    }

    public Decimal USPrice {
        get { return usPrice; } set { usPrice = value; }
    }

    public string Comment {
        get { return comment; } set { comment = value; }
    }

    public DateTime ShipDate {
        get { return shipDate; } set { shipDate = value; }
    }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append($"PartNumber: {partNumber}\n");
        sb.Append($"ProductName: {productName}\n");
        sb.Append($"Quantity: {quantity}\n");
        sb.Append($"USPrice: {usPrice}\n");
        if (comment != null)
            sb.Append($"Comment: {comment}\n");
        if (shipDate != DateTime.MinValue)
            sb.Append($"ShipDate: {shipDate:d}\n");
        return sb.ToString();
    }
}

class PurchaseOrder
{
    private string purchaseOrderNumber;
    private DateTime orderDate;
    private string comment;
    private List<Address> addresses;
    private List<PurchaseOrderItem> items;

    public string PurchaseOrderNumber {
        get { return purchaseOrderNumber; } set { purchaseOrderNumber = value; }
    }
```

```
    public DateTime OrderDate {
        get { return orderDate; } set { orderDate = value; }
    }

    public string Comment {
        get { return comment; } set { comment = value; }
    }

    public List<Address> Addresses {
        get { return addresses; } set { addresses = value; }
    }

    public List<PurchaseOrderItem> Items {
        get { return items; } set { items = value; }
    }

    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append($"PurchaseOrderNumber: {purchaseOrderNumber}\n");
        sb.Append($"OrderDate: {orderDate:d}\n");
        sb.Append("\n");
        sb.Append("Addresses\n");
        sb.Append("=====\n");
        foreach (Address address in addresses)
        {
            sb.Append(address);
            sb.Append("\n");
        }
        sb.Append("Items\n");
        sb.Append("=====\n");
        foreach (PurchaseOrderItem item in items)
        {
            sb.Append(item);
            sb.Append("\n");
        }
        return sb.ToString();
    }
}

class Program {
    public static void Main()
    {
        XElement po = XElement.Load("PurchaseOrder.xml");
        PurchaseOrder purchaseOrder = new PurchaseOrder {
            PurchaseOrderNumber = (string)po.Attribute("PurchaseOrderNumber"),
            OrderDate = (DateTime)po.Attribute("OrderDate"),
            Addresses = (
                            from a in po.Elements("Address")
                            select new Address {
                                AddressType = ((string)a.Attribute("Type") == "Shipping") ?
                                    Address.AddressUse.Shipping :
                                    Address.AddressUse.Billing,
                                Name = (string)a.Element("Name"),
                                Street = (string)a.Element("Street"),
                                City = (string)a.Element("City"),
                                State = (string)a.Element("State"),
                                Zip = (string)a.Element("Zip"),
                                Country = (string)a.Element("Country")
                            }
                        ).ToList(),
            Items = (
                        from i in po.Element("Items").Elements("Item")
                        select new PurchaseOrderItem {
                            PartNumber = (string)i.Attribute("PartNumber"),
                            ProductName = (string)i.Element("ProductName"),
                            Quantity = (int)i.Element("Quantity"),
                            USPrice = (Decimal)i.Element("USPrice"),
```

```
                                Comment = (string)i.Element("Comment"),
                                ShipDate = (i.Element("ShipDate") != null) ?
                                    (DateTime)i.Element("ShipDate") :
                                    DateTime.MinValue
                            }
                        ).ToList()
            };
            Console.WriteLine(purchaseOrder);
        }
    }
```

En este ejemplo, el resultado de la consulta se devuelve como un tipo IEnumerable<T> de `PurchaseOrderItem`. Los elementos de la clase `PurchaseOrder` son del tipo IEnumerable<T> de `PurchaseOrderItem`. El código usa el método de extensión ToList para crear una colección List<T> de los resultados de la consulta.

El ejemplo genera el siguiente resultado:

```
PurchaseOrderNumber: 99503
OrderDate: 10/20/1999

Addresses
=====
Type: Shipping
Name: Ellen Adams
Street: 123 Maple Street
City: Mill Valley
State: CA
Zip: 10999
Country: USA

Type: Billing
Name: Tai Yee
Street: 8 Oak Avenue
City: Old Town
State: PA
Zip: 95819
Country: USA

Items
=====
PartNumber: 872-AA
ProductName: Lawnmower
Quantity: 1
USPrice: 148.95
Comment: Confirm this is electric

PartNumber: 926-AA
ProductName: Baby Monitor
Quantity: 2
USPrice: 39.98
ShipDate: 5/21/1999
```

Vea también

- Select
- ToList

Procedimiento para proyectar un tipo anónimo (C#)

18/03/2020 • 3 minutes to read • Edit Online

En algunos casos quizás desee proyectar una consulta a un nuevo tipo, aunque sepa que solo utilizará este tipo durante un breve período. Crear un nuevo tipo solamente para usarlo en la proyección conlleva mucho trabajo adicional. En este caso un enfoque más eficiente consiste en proyectar en un tipo anónimo. Los tipos anónimos permiten definir una clase y, a continuación, declarar e inicializar un objeto de esa clase, sin dar un nombre a la clase.

Los tipos anónimos son la implementación de C# del concepto matemático de *tupla*. El término matemático tupla se originó de la secuencia único, doble, triple, cuádruple, quíntuple, n-tuple. Se refiere a una secuencia finita de objetos, cada uno de un tipo específico. A veces se denomina una lista de pares nombre/valor. Por ejemplo, el contenido de una dirección en el documento XML Archivo XML de ejemplo: Pedido de compra común (LINQ to XML) se podría expresar de la siguiente manera:

```
Name: Ellen Adams
Street: 123 Maple Street
City: Mill Valley
State: CA
Zip: 90952
Country: USA
```

Cuando se crea una instancia de un tipo anónimo, resulta práctico pensar en ella como si se creara una tupla de orden n. Si escribe una consulta que crea una tupla en la cláusula `select`, la consulta devuelve un `IEnumerable` de la tupla.

Ejemplo

En este ejemplo, la cláusula `select` proyecta un tipo anónimo. A continuación, el ejemplo utiliza `var` para crear el objeto `IEnumerable`. En el bucle `foreach`, la variable de iteración se convierte en una instancia del tipo anónimo creado en la expresión de consulta.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

```
XElement custOrd = XElement.Load("CustomersOrders.xml");
var custList =
    from el in custOrd.Element("Customers").Elements("Customer")
    select new {
        CustomerID = (string)el.Attribute("CustomerID"),
        CompanyName = (string)el.Element("CompanyName"),
        ContactName = (string)el.Element("ContactName")
    };
foreach (var cust in custList)
    Console.WriteLine("{0}:{1}:{2}", cust.CustomerID, cust.CompanyName, cust.ContactName);
```

Este código genera el siguiente resultado:

```
GREAL:Great Lakes Food Market:Howard Snyder
HUNGC:Hungry Coyote Import Store:Yoshi Latimer
LAZYK:Lazy K Kountry Store:John Steel
LETSS:Let's Stop N Shop:Jaime Yorres
```

Procedimiento para generar archivos de texto a partir de XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este ejemplo muestra cómo generar un archivo de valores separados por comas (CSV) a partir de un archivo XML.

Ejemplo

La versión en C# de este ejemplo utiliza sintaxis de métodos y el operador `Aggregate` para generar un archivo CSV a partir de un documento XML en una única expresión. Para obtener más información, vea Sintaxis de consulta y sintaxis de método en LINQ.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

```
XElement custOrd = XElement.Load("CustomersOrders.xml");
string csv =
    (from el in custOrd.Element("Customers").Elements("Customer")
    select
        String.Format("{0},{1},{2},{3},{4},{5},{6},{7},{8},{9}{10}",
            (string)el.Attribute("CustomerID"),
            (string)el.Element("CompanyName"),
            (string)el.Element("ContactName"),
            (string)el.Element("ContactTitle"),
            (string)el.Element("Phone"),
            (string)el.Element("FullAddress").Element("Address"),
            (string)el.Element("FullAddress").Element("City"),
            (string)el.Element("FullAddress").Element("Region"),
            (string)el.Element("FullAddress").Element("PostalCode"),
            (string)el.Element("FullAddress").Element("Country"),
            Environment.NewLine
        )
    )
    .Aggregate(
        new StringBuilder(),
        (sb, s) => sb.Append(s),
        sb => sb.ToString()
    );
Console.WriteLine(csv);
```

Este código genera el siguiente resultado:

```
GREAL,Great Lakes Food Market,Howard Snyder,Marketing Manager,(503) 555-7555,2732 Baker
Blvd.,Eugene,OR,97403,USA
HUNGC,Hungry Coyote Import Store,Yoshi Latimer,Sales Representative,(503) 555-6874,City Center Plaza 516 Main
St.,Elgin,OR,97827,USA
LAZYK,Lazy K Kountry Store,John Steel,Marketing Manager,(509) 555-7969,12 Orchestra Terrace,Walla
Walla,WA,99362,USA
LETSS,Let's Stop N Shop,Jaime Yorres,Owner,(415) 555-5938,87 Polk St. Suite 5,San Francisco,CA,94117,USA
```

Vea también

- Proyecciones y transformaciones (LINQ to XML) (C#)

Procedimiento para generar XML a partir de archivos CSV (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra el uso de Language Integrated Query (LINQ) y LINQ to XML para generar un archivo XML a partir de un archivo de valores separados por comas (CSV).

Ejemplo

El siguiente código realiza una consulta LINQ sobre una matriz de cadenas.

La consulta usa la cláusula `let` para dividir cada cadena en una matriz de campos.

```
// Create the text file.
string csvString = @"GREAL,Great Lakes Food Market,Howard Snyder,Marketing Manager,(503) 555-7555,2732 Baker
Blvd.,Eugene,OR,97403,USA
HUNGC,Hungry Coyote Import Store,Yoshi Latimer,Sales Representative,(503) 555-6874,City Center Plaza 516 Main
St.,Elgin,OR,97827,USA
LAZYK,Lazy K Kountry Store,John Steel,Marketing Manager,(509) 555-7969,12 Orchestra Terrace,Walla
Walla,WA,99362,USA
LETSS,Let's Stop N Shop,Jaime Yorres,Owner,(415) 555-5938,87 Polk St. Suite 5,San Francisco,CA,94117,USA";
File.WriteAllText("cust.csv", csvString);

// Read into an array of strings.
string[] source = File.ReadAllLines("cust.csv");
XElement cust = new XElement("Root",
    from str in source
    let fields = str.Split(',')
    select new XElement("Customer",
        new XAttribute("CustomerID", fields[0]),
        new XElement("CompanyName", fields[1]),
        new XElement("ContactName", fields[2]),
        new XElement("ContactTitle", fields[3]),
        new XElement("Phone", fields[4]),
        new XElement("FullAddress",
            new XElement("Address", fields[5]),
            new XElement("City", fields[6]),
            new XElement("Region", fields[7]),
            new XElement("PostalCode", fields[8]),
            new XElement("Country", fields[9])
        )
    )
);
Console.WriteLine(cust);
```

Este código genera el siguiente resultado:

```xml
<Root>
  <Customer CustomerID="GREAL">
    <CompanyName>Great Lakes Food Market</CompanyName>
    <ContactName>Howard Snyder</ContactName>
    <ContactTitle>Marketing Manager</ContactTitle>
    <Phone>(503) 555-7555</Phone>
    <FullAddress>
      <Address>2732 Baker Blvd.</Address>
      <City>Eugene</City>
      <Region>OR</Region>
      <PostalCode>97403</PostalCode>
      <Country>USA</Country>
    </FullAddress>
  </Customer>
  <Customer CustomerID="HUNGC">
    <CompanyName>Hungry Coyote Import Store</CompanyName>
    <ContactName>Yoshi Latimer</ContactName>
    <ContactTitle>Sales Representative</ContactTitle>
    <Phone>(503) 555-6874</Phone>
    <FullAddress>
      <Address>City Center Plaza 516 Main St.</Address>
      <City>Elgin</City>
      <Region>OR</Region>
      <PostalCode>97827</PostalCode>
      <Country>USA</Country>
    </FullAddress>
  </Customer>
  <Customer CustomerID="LAZYK">
    <CompanyName>Lazy K Kountry Store</CompanyName>
    <ContactName>John Steel</ContactName>
    <ContactTitle>Marketing Manager</ContactTitle>
    <Phone>(509) 555-7969</Phone>
    <FullAddress>
      <Address>12 Orchestra Terrace</Address>
      <City>Walla Walla</City>
      <Region>WA</Region>
      <PostalCode>99362</PostalCode>
      <Country>USA</Country>
    </FullAddress>
  </Customer>
  <Customer CustomerID="LETSS">
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <ContactTitle>Owner</ContactTitle>
    <Phone>(415) 555-5938</Phone>
    <FullAddress>
      <Address>87 Polk St. Suite 5</Address>
      <City>San Francisco</City>
      <Region>CA</Region>
      <PostalCode>94117</PostalCode>
      <Country>USA</Country>
    </FullAddress>
  </Customer>
</Root>
```

Procedimiento para combinar dos colecciones (LINQ to XML) (C#)

18/03/2020 • 4 minutes to read • Edit Online

A veces, un elemento o atributo de un documento XML puede hacer referencia a otro elemento o atributo. Por ejemplo, el documento XML Archivo XML de muestra: clientes y pedidos (LINQ to XML) contiene una lista de clientes y una lista de pedidos. Cada elemento `Customer` contiene un atributo `CustomerID`. Cada elemento `Order` contiene un elemento `CustomerID`. El elemento `CustomerID` de cada pedido hace referencia al atributo `CustomerID` de un cliente.

En el tema Sample XSD File: Customers and Orders (Archivo XSD de ejemplo: Clientes y pedidos) se incluye un XSD que se puede usar para validar este documento. Usa las características `xs:key` y `xs:keyref` de XSD para establecer que el atributo `CustomerID` del elemento `Customer` es una clave, y para establecer una relación entre el elemento `CustomerID` de cada elemento `Order` y el atributo `CustomerID` de cada elemento `Customer`.

Con LINQ to XML, puede aprovechar esta relación mediante la cláusula `join`.

Como no se dispone de ningún índice, la combinación tendrá un rendimiento en tiempo de ejecución bajo.

Para obtener más información sobre `join`, consulte Join Operations (C#) (Operaciones de combinación [C#]).

Ejemplo

El ejemplo siguiente combina los elementos `Customer` con los elementos `Order`, y genera un nuevo documento XML que incluye el elemento `CompanyName` en los pedidos.

Antes de ejecutar la consulta, el ejemplo valida que el documento cumple el esquema de Sample XSD File: Customers and Orders (Archivo XSD de ejemplo: Clientes y pedidos). Esto garantiza que la cláusula de combinación siempre funcionará.

Esta consulta recupera primero todos los elementos `Customer`, y luego los combina con los elementos `Order`. Selecciona sólo los pedidos de los clientes con un atributo `CustomerID` mayor que "K". A continuación, proyecta un nuevo elemento `Order` que contiene la información de cliente en cada pedido.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

En este ejemplo se usa el siguiente esquema XSD: Sample XSD File: Customers and Orders (Archivo XSD de ejemplo: Clientes y pedidos).

La combinación de este modo no funcionará bien. Las combinaciones se llevan a cabo mediante una búsqueda lineal. No hay tablas o índices hash disponibles para incrementar el rendimiento.

```
XmlSchemaSet schemas = new XmlSchemaSet();
schemas.Add("", "CustomersOrders.xsd");

Console.Write("Attempting to validate, ");
XDocument custOrdDoc = XDocument.Load("CustomersOrders.xml");

bool errors = false;
custOrdDoc.Validate(schemas, (o, e) =>
                    {
                        Console.WriteLine("{0}", e.Message);
                        errors = true;
                    });
Console.WriteLine("custOrdDoc {0}", errors ? "did not validate" : "validated");

if (!errors)
{
    // Join customers and orders, and create a new XML document with
    // a different shape.

    // The new document contains orders only for customers with a
    // CustomerID > 'K'
    XElement custOrd = custOrdDoc.Element("Root");
    XElement newCustOrd = new XElement("Root",
        from c in custOrd.Element("Customers").Elements("Customer")
        join o in custOrd.Element("Orders").Elements("Order")
                  on (string)c.Attribute("CustomerID") equals
                     (string)o.Element("CustomerID")
        where ((string)c.Attribute("CustomerID")).CompareTo("K") > 0
        select new XElement("Order",
            new XElement("CustomerID", (string)c.Attribute("CustomerID")),
            new XElement("CompanyName", (string)c.Element("CompanyName")),
            new XElement("ContactName", (string)c.Element("ContactName")),
            new XElement("EmployeeID", (string)o.Element("EmployeeID")),
            new XElement("OrderDate", (DateTime)o.Element("OrderDate"))
        )
    );
    Console.WriteLine(newCustOrd);
}
```

Este código genera el siguiente resultado:

```
Attempting to validate, custOrdDoc validated
<Root>
  <Order>
    <CustomerID>LAZYK</CustomerID>
    <CompanyName>Lazy K Kountry Store</CompanyName>
    <ContactName>John Steel</ContactName>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1997-03-21T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LAZYK</CustomerID>
    <CompanyName>Lazy K Kountry Store</CompanyName>
    <ContactName>John Steel</ContactName>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-05-22T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1997-06-25T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-10-27T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <EmployeeID>6</EmployeeID>
    <OrderDate>1997-11-10T00:00:00</OrderDate>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <CompanyName>Let's Stop N Shop</CompanyName>
    <ContactName>Jaime Yorres</ContactName>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1998-02-12T00:00:00</OrderDate>
  </Order>
</Root>
```

Procedimiento para crear una jerarquía mediante la agrupación (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este ejemplo muestra cómo agrupar datos y después generar XML basado en la agrupación.

Ejemplo

Este ejemplo agrupa primero los datos por categorías y, a continuación, genera un nuevo archivo XML en el que la jerarquía XML refleja la agrupación.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Datos numéricos (LINQ to XML).

```
XElement doc = XElement.Load("Data.xml");
var newData =
    new XElement("Root",
        from data in doc.Elements("Data")
        group data by (string)data.Element("Category") into groupedData
        select new XElement("Group",
            new XAttribute("ID", groupedData.Key),
            from g in groupedData
            select new XElement("Data",
                g.Element("Quantity"),
                g.Element("Price")
            )
        )
    );
Console.WriteLine(newData);
```

Este ejemplo produce el siguiente resultado:

```xml
<Root>
  <Group ID="A">
    <Data>
      <Quantity>3</Quantity>
      <Price>24.50</Price>
    </Data>
    <Data>
      <Quantity>5</Quantity>
      <Price>4.95</Price>
    </Data>
    <Data>
      <Quantity>3</Quantity>
      <Price>66.00</Price>
    </Data>
    <Data>
      <Quantity>15</Quantity>
      <Price>29.00</Price>
    </Data>
  </Group>
  <Group ID="B">
    <Data>
      <Quantity>1</Quantity>
      <Price>89.99</Price>
    </Data>
    <Data>
      <Quantity>10</Quantity>
      <Price>.99</Price>
    </Data>
    <Data>
      <Quantity>8</Quantity>
      <Price>6.99</Price>
    </Data>
  </Group>
</Root>
```

Procedimiento para consultar LINQ to XML mediante XPath (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema presenta los métodos de extensión que permiten consultar un árbol XML con XPath. Para obtener información detallada acerca del uso de estos métodos de extensión, vea System.Xml.XPath.Extensions.

A menos que tenga un motivo muy específico para realizar consultas con XPath, como en el caso del uso intensivo de código heredado, no se recomienda usar XPath con LINQ to XML. Las consultas XPath no se realizarán tan bien como las consultas LINQ to XML.

Ejemplo

En el ejemplo siguiente se crea un árbol XML pequeño y se utiliza XPathSelectElements para seleccionar un conjunto de elementos.

```
XElement root = new XElement("Root",
    new XElement("Child1", 1),
    new XElement("Child1", 2),
    new XElement("Child1", 3),
    new XElement("Child2", 4),
    new XElement("Child2", 5),
    new XElement("Child2", 6)
);
IEnumerable<XElement> list = root.XPathSelectElements("./Child2");
foreach (XElement el in list)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
<Child2>4</Child2>
<Child2>5</Child2>
<Child2>6</Child2>
```

Procedimiento para escribir un método de eje de LINQ to XML (C#)

29/06/2020 • 4 minutes to read • Edit Online

Un método de eje de XML recupera una colección de elementos XML de un documento o de un elemento antecesor XML. Puede escribir sus propios métodos de ejes para recuperar recopilaciones de un árbol XML. Una de las mejores formas de hacerlo es escribir un método de extensión que devuelva una recopilación de elementos o atributos. Puede escribir su método de extensión para devolver subconjuntos específicos de elementos o atributos, basándose en los requisitos de su aplicación.

Ejemplo

El siguiente ejemplo usa dos métodos de extensión. El primer método de extensión, `GetXPath` funciona en XObject y devuelve una expresión XPath que cuando se evalúa devuelve el nodo o el atributo. El segundo método de extensión, `Find`, funciona en XElement. Devuelve una recopilación de objetos XAttribute y XElement que contienen algún texto especificado.

Este ejemplo utiliza el siguiente documento XML: Archivo XML de ejemplo: Varios pedidos de compra (LINQ to XML).

```
public static class MyExtensions
{
    private static string GetQName(XElement xe)
    {
        string prefix = xe.GetPrefixOfNamespace(xe.Name.Namespace);
        if (xe.Name.Namespace == XNamespace.None || prefix == null)
            return xe.Name.LocalName.ToString();
        else
            return prefix + ":" + xe.Name.LocalName.ToString();
    }

    private static string GetQName(XAttribute xa)
    {
        string prefix =
            xa.Parent.GetPrefixOfNamespace(xa.Name.Namespace);
        if (xa.Name.Namespace == XNamespace.None || prefix == null)
            return xa.Name.ToString();
        else
            return prefix + ":" + xa.Name.LocalName;
    }

    private static string NameWithPredicate(XElement el)
    {
        if (el.Parent != null && el.Parent.Elements(el.Name).Count() != 1)
            return GetQName(el) + "[" +
                (el.ElementsBeforeSelf(el.Name).Count() + 1) + "]";
        else
            return GetQName(el);
    }

    public static string StrCat<T>(this IEnumerable<T> source,
        string separator)
    {
        return source.Aggregate(new StringBuilder(),
                    (sb, i) => sb
                        .Append(i.ToString())
                        .Append(separator),
```

```
                s => s.ToString());
    }

    public static string GetXPath(this XObject xobj)
    {
        if (xobj.Parent == null)
        {
            XDocument doc = xobj as XDocument;
            if (doc != null)
                return ".";
            XElement el = xobj as XElement;
            if (el != null)
                return "/" + NameWithPredicate(el);
            // the XPath data model does not include white space text nodes
            // that are children of a document, so this method returns null.
            XText xt = xobj as XText;
            if (xt != null)
                return null;
            XComment com = xobj as XComment;
            if (com != null)
                return
                    "/" +
                    (
                        com
                        .Document
                        .Nodes()
                        .OfType<XComment>()
                        .Count() != 1 ?
                        "comment()[" +
                        (com
                        .NodesBeforeSelf()
                        .OfType<XComment>()
                        .Count() + 1) +
                        "]" :
                        "comment()"
                    );
            XProcessingInstruction pi = xobj as XProcessingInstruction;
            if (pi != null)
                return
                    "/" +
                    (
                        pi.Document.Nodes()
                        .OfType<XProcessingInstruction>()
                        .Count() != 1 ?
                        "processing-instruction()[" +
                        (pi
                        .NodesBeforeSelf()
                        .OfType<XProcessingInstruction>()
                        .Count() + 1) +
                        "]" :
                        "processing-instruction()"
                    );
            return null;
        }
        else
        {
            XElement el = xobj as XElement;
            if (el != null)
            {
                return
                    "/" +
                    el
                    .Ancestors()
                    .InDocumentOrder()
                    .Select(e => NameWithPredicate(e))
                    .StrCat("/") +
                    NameWithPredicate(el);
            }
            XAttribute at = xobj as XAttribute;
```

```
            if (at != null)
                return
                    "/" +
                    at
                    .Parent
                    .AncestorsAndSelf()
                    .InDocumentOrder()
                    .Select(e => NameWithPredicate(e))
                    .StrCat("/") +
                    "@" + GetQName(at);
            XComment com = xobj as XComment;
            if (com != null)
                return
                    "/" +
                    com
                    .Parent
                    .AncestorsAndSelf()
                    .InDocumentOrder()
                    .Select(e => NameWithPredicate(e))
                    .StrCat("/") +
                    (
                        com
                        .Parent
                        .Nodes()
                        .OfType<XComment>()
                        .Count() != 1 ?
                        "comment()[" +
                        (com
                        .NodesBeforeSelf()
                        .OfType<XComment>()
                        .Count() + 1) + "]" :
                        "comment()"
                    );
            XCData cd = xobj as XCData;
            if (cd != null)
                return
                    "/" +
                    cd
                    .Parent
                    .AncestorsAndSelf()
                    .InDocumentOrder()
                    .Select(e => NameWithPredicate(e))
                    .StrCat("/") +
                    (
                        cd
                        .Parent
                        .Nodes()
                        .OfType<XText>()
                        .Count() != 1 ?
                        "text()[" +
                        (cd
                        .NodesBeforeSelf()
                        .OfType<XText>()
                        .Count() + 1) + "]" :
                        "text()"
                    );
            XText tx = xobj as XText;
            if (tx != null)
                return
                    "/" +
                    tx
                    .Parent
                    .AncestorsAndSelf()
                    .InDocumentOrder()
                    .Select(e => NameWithPredicate(e))
                    .StrCat("/") +
                    (
                        tx
                        .Parent
```

```
                        .Nodes()
                        .OfType<XText>()
                        .Count() != 1 ?
                        "text()[" +
                        (tx
                        .NodesBeforeSelf()
                        .OfType<XText>()
                        .Count() + 1) + "]" :
                        "text()"
                    );
            XProcessingInstruction pi = xobj as XProcessingInstruction;
            if (pi != null)
                return
                    "/" +
                    pi
                    .Parent
                    .AncestorsAndSelf()
                    .InDocumentOrder()
                    .Select(e => NameWithPredicate(e))
                    .StrCat("/") +
                    (
                        pi
                        .Parent
                        .Nodes()
                        .OfType<XProcessingInstruction>()
                        .Count() != 1 ?
                        "processing-instruction()[" +
                        (pi
                        .NodesBeforeSelf()
                        .OfType<XProcessingInstruction>()
                        .Count() + 1) + "]" :
                        "processing-instruction()"
                    );
            return null;
        }
    }

    public static IEnumerable<XObject> Find(this XElement source, string value)
    {
        if (source.Attributes().Any())
        {
            foreach (XAttribute att in source.Attributes())
            {
                string contents = (string)att;
                if (contents.Contains(value))
                    yield return att;
            }
        }
        if (source.Elements().Any())
        {
            foreach (XElement child in source.Elements())
                foreach (XObject s in child.Find(value))
                    yield return s;
        }
        else
        {
            string contents = (string)source;
            if (contents.Contains(value))
                yield return source;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        XElement purchaseOrders = XElement.Load("PurchaseOrders.xml");
```

```
        IEnumerable<XObject> subset =
            from xobj in purchaseOrders.Find("1999")
            select xobj;

        foreach (XObject obj in subset)
        {
            Console.WriteLine(obj.GetXPath());
            if (obj.GetType() == typeof(XElement))
                Console.WriteLine(((XElement)obj).Value);
            else if (obj.GetType() == typeof(XAttribute))
                Console.WriteLine(((XAttribute)obj).Value);
        }
    }
}
```

Este código genera el siguiente resultado:

```
/PurchaseOrders/PurchaseOrder[1]/@OrderDate
1999-10-20
/PurchaseOrders/PurchaseOrder[1]/Items/Item[2]/ShipDate
1999-05-21
/PurchaseOrders/PurchaseOrder[2]/@OrderDate
1999-10-22
/PurchaseOrders/PurchaseOrder[3]/@OrderDate
1999-10-22
```

Procedimiento para efectuar transformaciones en streaming de texto en XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

Un enfoque del procesamiento de un archivo de texto es escribir un método de extensión que transmita el archivo de texto por secuencias de línea en línea mediante la construcción `yield return`. Después, puede escribir una consulta LINQ que procese el archivo de texto de forma aplazada y lenta. Si después usa XStreamingElement para transmitir el resultado, puede crear una transformación del archivo de texto al XML usando una cantidad mínima de memoria, independientemente del tamaño del archivo de texto de origen.

Existen algunas advertencias que deben tenerse en cuenta sobre las transformaciones de transmisión por secuencias. Una transformación de transmisión por secuencias se aplica mejor en situaciones en las que se puede procesar todo el archivo, y si se pueden procesar las líneas en el orden en el que se producen en el documento de origen. Si se debe procesar más de un archivo simultáneamente so si se deben ordenar las líneas antes de poder procesarlas, se perderán muchas de las ventajas de utilizar una técnica de transmisión por secuencias.

Ejemplo

El siguiente archivo de texto, People.txt, es el origen de este ejemplo.

```
#This is a comment
1,Tai,Yee,Writer
2,Nikolay,Grachev,Programmer
3,David,Wright,Inventor
```

El siguiente código contiene un método de extensión que transmite las líneas del archivo de texto por secuencias de forma diferida.

```
public static class StreamReaderSequence
{
    public static IEnumerable<string> Lines(this StreamReader source)
    {
        if (source == null)
            throw new ArgumentNullException(nameof(source));

        string line;
        while ((line = source.ReadLine()) != null)
        {
            yield return line;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        var sr = new StreamReader("People.txt");
        var xmlTree = new XStreamingElement("Root",
            from line in sr.Lines()
            let items = line.Split(',')
            where !line.StartsWith("#")
            select new XElement("Person",
                       new XAttribute("ID", items[0]),
                       new XElement("First", items[1]),
                       new XElement("Last", items[2]),
                       new XElement("Occupation", items[3])
                   )
        );
        Console.WriteLine(xmlTree);
        sr.Close();
    }
}
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <Person ID="1">
    <First>Tai</First>
    <Last>Yee</Last>
    <Occupation>Writer</Occupation>
  </Person>
  <Person ID="2">
    <First>Nikolay</First>
    <Last>Grachev</Last>
    <Occupation>Programmer</Occupation>
  </Person>
  <Person ID="3">
    <First>David</First>
    <Last>Wright</Last>
    <Occupation>Inventor</Occupation>
  </Person>
</Root>
```

Vea también

- XStreamingElement

Procedimiento para mostrar todos los nodos de un árbol (C#)

18/03/2020 • 5 minutes to read • Edit Online

A veces resulta útil enumerar todos los nodos de un árbol. Puede resultar útil cuando se aprende exactamente cómo afecta al árbol un método o propiedad. Un enfoque para enumerar todos los nodos en formato de texto consiste en generar una expresión XPath que identifique de forma exacta y específica cualquier nodo del árbol.

No resulta particularmente útil ejecutar expresiones XPath mediante LINQ to XML. Las expresiones XPath tienen un rendimiento menor que las consultas de LINQ to XML y las consultas de LINQ to XML son mucho más eficaces. No obstante, como forma de identificar nodos en el árbol XML, XPath funciona bien.

Ejemplo

En este ejemplo se muestra una función con el nombre `GetXPath` que genera una expresión XPath específica para cualquier nodo del árbol XML. Genera expresiones XPath adecuadas incluso cuando hay nodos en un espacio de nombres. Las expresiones XPath se generan usando prefijos de espacio de nombres.

A continuación el ejemplo crea un pequeño árbol de texto que contiene un ejemplo de varios tipos de nodos. Después recorre en iteración los nodos descendientes y muestra la expresión XPath para cada nodo.

Observará que la declaración XML no es un nodo del árbol.

A continuación se muestra un archivo XML que contiene varios tipos de nodos:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<?target data?>
<Root AttName="An Attribute" xmlns:aw="http://www.adventure-works.com">
  <!--This is a comment-->
  <Child>Text</Child>
  <Child>Other Text</Child>
  <ChildWithMixedContent>text<b>BoldText</b>otherText</ChildWithMixedContent>
  <aw:ElementInNamespace>
    <aw:ChildInNamespace />
  </aw:ElementInNamespace>
</Root>
```

A continuación se muestra la lista de nodos del árbol XML anterior, expresados como expresiones XPath:

```
/processing-instruction()
/Root
/Root/@AttName
/Root/@xmlns:aw
/Root/comment()
/Root/Child[1]
/Root/Child[1]/text()
/Root/Child[2]
/Root/Child[2]/text()
/Root/ChildWithMixedContent
/Root/ChildWithMixedContent/text()[1]
/Root/ChildWithMixedContent/b
/Root/ChildWithMixedContent/b/text()
/Root/ChildWithMixedContent/text()[2]
/Root/aw:ElementInNamespace
/Root/aw:ElementInNamespace/aw:ChildInNamespace
```

```
public static class MyExtensions
{
    private static string GetQName(XElement xe)
    {
        string prefix = xe.GetPrefixOfNamespace(xe.Name.Namespace);
        if (xe.Name.Namespace == XNamespace.None || prefix == null)
            return xe.Name.LocalName.ToString();
        else
            return prefix + ":" + xe.Name.LocalName.ToString();
    }

    private static string GetQName(XAttribute xa)
    {
        string prefix =
            xa.Parent.GetPrefixOfNamespace(xa.Name.Namespace);
        if (xa.Name.Namespace == XNamespace.None || prefix == null)
            return xa.Name.ToString();
        else
            return prefix + ":" + xa.Name.LocalName;
    }

    private static string NameWithPredicate(XElement el)
    {
        if (el.Parent != null && el.Parent.Elements(el.Name).Count() != 1)
            return GetQName(el) + "[" +
                (el.ElementsBeforeSelf(el.Name).Count() + 1) + "]";
        else
            return GetQName(el);
    }

    public static string StrCat<T>(this IEnumerable<T> source,
        string separator)
    {
        return source.Aggregate(new StringBuilder(),
                    (sb, i) => sb
                        .Append(i.ToString())
                        .Append(separator),
                    s => s.ToString());
    }

    public static string GetXPath(this XObject xobj)
    {
        if (xobj.Parent == null)
        {
            XDocument doc = xobj as XDocument;
            if (doc != null)
                return ".";
            XElement el = xobj as XElement;
            if (el != null)
```

```
            return "/" + NameWithPredicate(el);
        // the XPath data model does not include white space text nodes
        // that are children of a document, so this method returns null.
        XText xt = xobj as XText;
        if (xt != null)
            return null;
        XComment com = xobj as XComment;
        if (com != null)
            return
                "/" +
                (
                    com
                    .Document
                    .Nodes()
                    .OfType<XComment>()
                    .Count() != 1 ?
                    "comment()[" +
                    (com
                    .NodesBeforeSelf()
                    .OfType<XComment>()
                    .Count() + 1) +
                    "]" :
                    "comment()"
                );
        XProcessingInstruction pi = xobj as XProcessingInstruction;
        if (pi != null)
            return
                "/" +
                (
                    pi.Document.Nodes()
                    .OfType<XProcessingInstruction>()
                    .Count() != 1 ?
                    "processing-instruction()[" +
                    (pi
                    .NodesBeforeSelf()
                    .OfType<XProcessingInstruction>()
                    .Count() + 1) +
                    "]" :
                    "processing-instruction()"
                );
        return null;
    }
    else
    {
        XElement el = xobj as XElement;
        if (el != null)
        {
            return
                "/" +
                el
                .Ancestors()
                .InDocumentOrder()
                .Select(e => NameWithPredicate(e))
                .StrCat("/") +
                NameWithPredicate(el);
        }
        XAttribute at = xobj as XAttribute;
        if (at != null)
            return
                "/" +
                at
                .Parent
                .AncestorsAndSelf()
                .InDocumentOrder()
                .Select(e => NameWithPredicate(e))
                .StrCat("/") +
                "@" + GetQName(at);
        XComment com = xobj as XComment;
        if (com != null)
```

```
            return
                "/" +
                com
                .Parent
                .AncestorsAndSelf()
                .InDocumentOrder()
                .Select(e => NameWithPredicate(e))
                .StrCat("/") +
                (
                    com
                    .Parent
                    .Nodes()
                    .OfType<XComment>()
                    .Count() != 1 ?
                    "comment()[" +
                    (com
                    .NodesBeforeSelf()
                    .OfType<XComment>()
                    .Count() + 1) + "]" :
                    "comment()"
                );
        XCData cd = xobj as XCData;
        if (cd != null)
            return
                "/" +
                cd
                .Parent
                .AncestorsAndSelf()
                .InDocumentOrder()
                .Select(e => NameWithPredicate(e))
                .StrCat("/") +
                (
                    cd
                    .Parent
                    .Nodes()
                    .OfType<XText>()
                    .Count() != 1 ?
                    "text()[" +
                    (cd
                    .NodesBeforeSelf()
                    .OfType<XText>()
                    .Count() + 1) + "]" :
                    "text()"
                );
        XText tx = xobj as XText;
        if (tx != null)
            return
                "/" +
                tx
                .Parent
                .AncestorsAndSelf()
                .InDocumentOrder()
                .Select(e => NameWithPredicate(e))
                .StrCat("/") +
                (
                    tx
                    .Parent
                    .Nodes()
                    .OfType<XText>()
                    .Count() != 1 ?
                    "text()[" +
                    (tx
                    .NodesBeforeSelf()
                    .OfType<XText>()
                    .Count() + 1) + "]" :
                    "text()"
                );
        XProcessingInstruction pi = xobj as XProcessingInstruction;
        if (pi != null)
```

```
                return
                    "/" +
                    pi
                    .Parent
                    .AncestorsAndSelf()
                    .InDocumentOrder()
                    .Select(e => NameWithPredicate(e))
                    .StrCat("/") +
                    (
                        pi
                        .Parent
                        .Nodes()
                        .OfType<XProcessingInstruction>()
                        .Count() != 1 ?
                        "processing-instruction()[" +
                        (pi
                        .NodesBeforeSelf()
                        .OfType<XProcessingInstruction>()
                        .Count() + 1) + "]" :
                        "processing-instruction()"
                    );
            return null;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        XNamespace aw = "http://www.adventure-works.com";
        XDocument doc = new XDocument(
            new XDeclaration("1.0", "utf-8", "yes"),
            new XProcessingInstruction("target", "data"),
            new XElement("Root",
                new XAttribute("AttName", "An Attribute"),
                new XAttribute(XNamespace.Xmlns + "aw", aw.ToString()),
                new XComment("This is a comment"),
                new XElement("Child",
                    new XText("Text")
                ),
                new XElement("Child",
                    new XText("Other Text")
                ),
                new XElement("ChildWithMixedContent",
                    new XText("text"),
                    new XElement("b", "BoldText"),
                    new XText("otherText")
                ),
                new XElement(aw + "ElementInNamespace",
                    new XElement(aw + "ChildInNamespace")
                )
            )
        );
        doc.Save("Test.xml");
        Console.WriteLine(File.ReadAllText("Test.xml"));
        Console.WriteLine("------");
        foreach (XObject obj in doc.DescendantNodes())
        {
            Console.WriteLine(obj.GetXPath());
            XElement el = obj as XElement;
            if (el != null)
                foreach (XAttribute at in el.Attributes())
                    Console.WriteLine(at.GetXPath());
        }
    }
}
```

Este ejemplo produce el siguiente resultado:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<?target data?>
<Root AttName="An Attribute" xmlns:aw="http://www.adventure-works.com">
  <!--This is a comment-->
  <Child>Text</Child>
  <Child>Other Text</Child>
  <ChildWithMixedContent>text<b>BoldText</b>otherText</ChildWithMixedContent>
  <aw:ElementInNamespace>
    <aw:ChildInNamespace />
  </aw:ElementInNamespace>
</Root>
------
/processing-instruction()
/Root
/Root/@AttName
/Root/@xmlns:aw
/Root/comment()
/Root/Child[1]
/Root/Child[1]/text()
/Root/Child[2]
/Root/Child[2]/text()
/Root/ChildWithMixedContent
/Root/ChildWithMixedContent/text()[1]
/Root/ChildWithMixedContent/b
/Root/ChildWithMixedContent/b/text()
/Root/ChildWithMixedContent/text()[2]
/Root/aw:ElementInNamespace
/Root/aw:ElementInNamespace/aw:ChildInNamespace
```

Procedimiento para recuperar párrafos de un documento XML abierto de Office (C#)

18/03/2020 • 3 minutes to read • Edit Online

Este tema presenta un ejemplo que abre un documento XML abierto de Office y recupera una colección de todos los párrafos del documento.

Para más información sobre Office Open XML, vea el SDK de Open XML y www.ericwhite.com.

Ejemplo

Este ejemplo abre un paquete XML abierto de Office y usa las relaciones de dicho paquete para buscar las partes del documento y del estilo. A continuación, consulta el documento y proyecta una colección de un tipo anónimo que contiene el nodo XElement del párrafo, así como el nombre de estilo y el texto de cada párrafo.

El ejemplo usa un método de extensión denominado `StringConcatenate`, también proporcionado.

Para ver un tutorial en el que se explica cómo funciona este ejemplo, consulte Transformaciones funcionales puras de XML (C#).

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
```

```
}

class Program
{
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
               .Elements(w + "r")
               .Elements(w + "t")
               .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
          "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
              wdPackage
              .GetRelationshipsByType(documentRelationshipType)
              .FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri =
                    PackUriHelper
                    .ResolvePartUri(
                       new Uri("/", UriKind.Relative),
                             docPackageRelationship.TargetUri);
                PackagePart documentPart =
                    wdPackage.GetPart(documentUri);

                //  Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                //  Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                  documentPart.GetRelationshipsByType(stylesRelationshipType)
                  .FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    //  Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                      (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");
```

```
        // Find all paragraphs in the document.
        var paragraphs =
            from para in xDoc
                         .Root
                         .Element(w + "body")
                         .Descendants(w + "p")
            let styleNode = para
                            .Elements(w + "pPr")
                            .Elements(w + "pStyle")
                            .FirstOrDefault()
            select new
            {
                ParagraphNode = para,
                StyleName = styleNode != null ?
                    (string)styleNode.Attribute(w + "val") :
                    defaultStyle
            };

        // Retrieve the text of each paragraph.
        var paraWithText =
            from para in paragraphs
            select new
            {
                ParagraphNode = para.ParagraphNode,
                StyleName = para.StyleName,
                Text = ParagraphText(para.ParagraphNode)
            };

        foreach (var p in paraWithText)
            Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
    }
}
```

En este ejemplo se genera el resultado siguiente cuando se ejecuta con el documento descrito en Crear el documento de origen de Office Open XML (C#):

```
StyleName:Heading1 >Parsing WordprocessingML with LINQ to XML<
StyleName:Normal ><
StyleName:Normal >The following example prints to the console.<
StyleName:Normal ><
StyleName:Code >using System;<
StyleName:Code ><
StyleName:Code >class Program {<
StyleName:Code >    public static void (string[] args) {<
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >    }<
StyleName:Code >}<
StyleName:Normal ><
StyleName:Normal >This example produces the following output:<
StyleName:Normal ><
StyleName:Code >Hello World<
```

Procedimiento para modificar un documento XML abierto de Office (C#)

18/03/2020 • 3 minutes to read • Edit Online

En este tema se presenta un ejemplo que abre un documento XML abierto de Office, lo modifica y lo guarda.

Para más información sobre Office Open XML, vea el SDK de Open XML y www.ericwhite.com.

Ejemplo

Este ejemplo busca el primer elemento de párrafo del documento. Recupera el texto del párrafo y, a continuación, elimina todas las ejecuciones de texto del párrafo. Crea una nueva ejecución de texto que está formada por el texto del primer párrafo que se ha convertido a mayúsculas. A continuación serializa el XML cambiado en el paquete de XML abierto y lo cierra.

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
    public static string ParagraphText(XElement e)
    {
```

```
    {
        XNamespace w = e.Name.Namespace;
        return e
                .Elements(w + "r")
                .Elements(w + "t")
                .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
          "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.ReadWrite))
        {
            PackageRelationship docPackageRelationship =
              wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
                  docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);

                //  Load the document XML in the part into an XDocument instance.
                XDocument xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                //  Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                  documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                PackagePart stylePart = null;
                XDocument styleDoc = null;

                if (styleRelation != null)
                {
                    Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    stylePart = wdPackage.GetPart(styleUri);

                    //  Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }

                XElement paraNode = xDoc
                                    .Root
                                    .Element(w + "body")
                                    .Descendants(w + "p")
                                    .FirstOrDefault();

                string paraText = ParagraphText(paraNode);

                // remove all text runs
                paraNode.Descendants(w + "r").Remove();

                paraNode.Add(
                    new XElement(w + "r",
                        new XElement(w + "t", paraText.ToUpper())
                    )
                );

                //  Save the XML into the package
                using (XmlWriter xw =
                  XmlWriter.Create(documentPart.GetStream(FileMode.Create, FileAccess.Write)))
                {
```

```
                    xDoc.Save(xw);
                }

                Console.WriteLine("New first paragraph: >{0}<", paraText.ToUpper());
            }
        }
    }
}
```

Si abre `SampleDoc.docx` tras ejecutar este programa, puede ver que este programa ha convertido el primer párrafo del documento a mayúsculas.

En este ejemplo se genera el resultado siguiente cuando se ejecuta con el documento descrito en Crear el documento de origen de Office Open XML (C#):

```
New first paragraph: >PARSING WORDPROCESSINGML WITH LINQ TO XML<
```

Procedimiento para rellenar un árbol XML desde el sistema de archivos (C#)

18/03/2020 • 2 minutes to read • Edit Online

Una aplicación habitual y útil de los árboles XML es un almacén de datos de nombres y valores jerárquicos. Puede rellenar un árbol XML con datos jerárquicos y, a continuación, consultarlo, transformarlo y, si es necesario, serializarlo. En este escenario de uso, gran parte de la semántica específica XML (por ejemplo, el comportamiento de los espacios en blanco y los espacios de nombres) no es importante. En su lugar, usará el árbol XML como una base de datos jerárquica, pequeña y en memoria, de usuario único.

Ejemplo

El ejemplo siguiente rellena un árbol XML desde el sistema de archivos local mediante la recursividad. A continuación, consulta el árbol y calcula el total de los tamaños de todos los archivos del árbol.

```
class Program
{
    static XElement CreateFileSystemXmlTree(string source)
    {
        DirectoryInfo di = new DirectoryInfo(source);
        return new XElement("Dir",
            new XAttribute("Name", di.Name),
            from d in Directory.GetDirectories(source)
            select CreateFileSystemXmlTree(d),
            from fi in di.GetFiles()
            select new XElement("File",
                new XElement("Name", fi.Name),
                new XElement("Length", fi.Length)
            )
        );
    }

    static void Main(string[] args)
    {
        XElement fileSystemTree = CreateFileSystemXmlTree("C:/Tmp");
        Console.WriteLine(fileSystemTree);
        Console.WriteLine("------");
        long totalFileSize =
            (from f in fileSystemTree.Descendants("File")
             select (long)f.Element("Length")).Sum();
        Console.WriteLine("Total File Size:{0}", totalFileSize);
    }
}
```

Este ejemplo genera un resultado similar al siguiente:

```
<Dir Name="Tmp">
  <Dir Name="ConsoleApplication1">
    <Dir Name="bin">
      <Dir Name="Debug">
        <File>
          <Name>ConsoleApplication1.exe</Name>
          <Length>4608</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.pdb</Name>
          <Length>11776</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.vshost.exe</Name>
          <Length>9568</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.vshost.exe.manifest</Name>
          <Length>473</Length>
        </File>
      </Dir>
    </Dir>
    <Dir Name="obj">
      <Dir Name="Debug">
        <Dir Name="TempPE" />
        <File>
          <Name>ConsoleApplication1.csproj.FileListAbsolute.txt</Name>
          <Length>322</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.exe</Name>
          <Length>4608</Length>
        </File>
        <File>
          <Name>ConsoleApplication1.pdb</Name>
          <Length>11776</Length>
        </File>
      </Dir>
    </Dir>
    <Dir Name="Properties">
      <File>
        <Name>AssemblyInfo.cs</Name>
        <Length>1454</Length>
      </File>
    </Dir>
    <File>
      <Name>ConsoleApplication1.csproj</Name>
      <Length>2546</Length>
    </File>
    <File>
      <Name>ConsoleApplication1.sln</Name>
      <Length>937</Length>
    </File>
    <File>
      <Name>ConsoleApplication1.suo</Name>
      <Length>10752</Length>
    </File>
    <File>
      <Name>Program.cs</Name>
      <Length>269</Length>
    </File>
  </Dir>
</Dir>
------
Total File Size:59089
```

Comparación de XPath y LINQ to XML

18/03/2020 • 6 minutes to read • Edit Online

XPath y LINQ to XML ofrecen una funcionalidad similar. Ambos se pueden usar para consultar un árbol XML, devolviendo resultados como recopilación de elementos, recopilación de nodos o el valor de un elemento o atributo. Sin embargo, también hay algunas diferencias.

Diferencias entre XPath y LINQ to XML

XPath no permite la proyección de nuevos tipos. Solo puede devolver recopilaciones de nodos del árbol, mientras que LINQ to XML puede ejecutar una consulta y proyectar un gráfico de objetos o un árbol XML en una nueva forma. Las consultas de LINQ to XML abarcan una funcionalidad mucho mayor y son mucho más eficaces que las expresiones XPath.

Las expresiones XPath existen de forma aislada en una cadena. El compilador de C# no puede ayudar a analizar la expresión XPath en tiempo de compilación. Por el contrario, el compilador de C# analiza y compila las consultas de LINQ to XML. El compilador puede detectar muchos errores de consulta.

Los resultados de XPath no están fuertemente tipados. En varias circunstancias, el resultado de evaluar una expresión XPath es un objeto, y depende del desarrollador determinar el tipo adecuado y convertir el resultado según sea necesario. Por el contrario, las proyecciones de una consulta de LINQ to XML están fuertemente tipadas.

Ordenación de resultados

La recomendación XPath 1.0 indica que una recopilación que es el resultado de evaluar una expresión XPath está desordenada.

Pero cuando se recorre en iteración una recopilación devuelta por un método de eje XPath de LINQ to XML, los nodos de la recopilación se devuelven en el orden del documento. Esto es aplicable incluso cuando se tiene acceso a los ejes XPath donde se expresan los predicados en términos de orden de documento inverso, como `preceding` y `preceding-sibling`.

Por el contrario, la mayoría de los ejes de LINQ to XML devuelven recopilaciones en el orden del documento, pero dos de ellos, Ancestors y AncestorsAndSelf, devuelven recopilaciones en el orden inverso del documento. En la siguiente tabla se enumeran los ejes y se indica el orden de la recopilación para cada uno:

EJE LINQ TO XML	ORDENACIÓN
XContainer.DescendantNodes	Orden del documento
XContainer.Descendants	Orden del documento
XContainer.Elements	Orden del documento
XContainer.Nodes	Orden del documento
XContainer.NodesAfterSelf	Orden del documento
XContainer.NodesBeforeSelf	Orden del documento

EJE LINQ TO XML	ORDENACIÓN
XElement.AncestorsAndSelf	Orden del documento inverso
XElement.Attributes	Orden del documento
XElement.DescendantNodesAndSelf	Orden del documento
XElement.DescendantsAndSelf	Orden del documento
XNode.Ancestors	Orden del documento inverso
XNode.ElementsAfterSelf	Orden del documento
XNode.ElementsBeforeSelf	Orden del documento
XNode.NodesAfterSelf	Orden del documento
XNode.NodesBeforeSelf	Orden del documento

Predicados de posición

En una expresión XPath, los predicados de posición se expresan en términos de orden del documento para muchos ejes, pero se expresan en el orden del documento inverso para los ejes inversos, que son `preceding`, `preceding-sibling`, `ancestor` y `ancestor-or-self`. Por ejemplo, la expresión de XPath `preceding-sibling::*[1]` devuelve el elemento del mismo nivel inmediatamente anterior. Esto es aplicable aunque el conjunto de resultados finales se presenta en el orden del documento.

Por el contrario, todos los predicados de posición de LINQ to XML siempre se expresan en términos del orden del eje. Por ejemplo, `anElement.ElementsBeforeSelf().ElementAt(0)` devuelve el primer elemento secundario del elemento primario del elemento consultado, no el elemento del mismo nivel inmediatamente anterior. Otro ejemplo: `anElement.Ancestors().ElementAt(0)` devuelve el elemento primario.

Si quisiera buscar el elemento inmediatamente precedente de LINQ to XML escribiría la siguiente expresión:

```
ElementsBeforeSelf().Last()
```


```
ElementsBeforeSelf().Last()
```


Diferencias de rendimiento

Las consultas XPath que usan la funcionalidad de XPath en LINQ to XML no tienen un rendimiento tan bueno como las consultas de LINQ to XML.

Comparación de la composición

La composición de una consulta de LINQ to XML es en cierta medida paralela a la composición de una expresión XPath, aunque tiene una sintaxis muy diferente.

Por ejemplo, si tiene un elemento de una variable con el nombre `customers` y desea buscar un elemento inferior con el nombre `CompanyName` bajo todos los elementos secundarios con el nombre `Customer`, debe escribir una

expresión XPath de la siguiente manera:

```
customers.XPathSelectElements("./Customer/CompanyName")
```


```
customers.XPathSelectElements("./Customer/CompanyName")
```


La consulta equivalente de LINQ to XML es la siguiente:

```
customers.Elements("Customer").Elements("CompanyName")
```


```
customers.Elements("Customer").Elements("CompanyName")
```


Existen elementos similares para cada uno de los ejes XPath.

EJE XPATH	EJE LINQ TO XML
secundario (el eje predeterminado)	XContainer.Elements
Primario (..)	XObject.Parent
eje de atributo (@)	XElement.Attribute o XElement.Attributes
eje antecesor	XNode.Ancestors
eje antecesor o propio	XElement.AncestorsAndSelf
eje descendiente (//)	XContainer.Descendants o XContainer.DescendantNodes
descendiente o propio	XElement.DescendantsAndSelf o XElement.DescendantNodesAndSelf
siguientes-relacionados	XNode.ElementsAfterSelf o XNode.NodesAfterSelf
precedentes-relacionados	XNode.ElementsBeforeSelf o XNode.NodesBeforeSelf

EJE XPATH	EJE LINQ TO XML
siguientes	No hay equivalente directo.
precedentes	No hay equivalente directo.

Procedimiento para buscar un elemento secundario (XPath-LINQ to XML) (C#)

19/03/2020 • 2 minutes to read • Edit Online

En este tema se compara el eje del elemento secundario XPath con el método LINQ to XML Element.

La expresión XPath es `DeliveryNotes`.

Ejemplo

Este ejemplo busca el elemento secundario `DeliveryNotes`.

Este ejemplo utiliza el siguiente documento XML: Archivo XML de ejemplo: Varios pedidos de compra (LINQ to XML).

```
XDocument cpo = XDocument.Load("PurchaseOrders.xml");
XElement po = cpo.Root.Element("PurchaseOrder");

// LINQ to XML query
XElement el1 = po.Element("DeliveryNotes");

// XPath expression
XElement el2 = po.XPathSelectElement("DeliveryNotes");
// same as "child::DeliveryNotes"
// same as "./DeliveryNotes"

if (el1 == el2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(el1);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
```

Procedimiento para buscar una lista de elementos secundarios (XPath-LINQ to XML) (C#)

19/03/2020 • 2 minutes to read • Edit Online

En este tema se compara el eje de los elementos secundarios de XPath con el eje LINQ to XML Elements.

La expresión XPath es: `./*`

Ejemplo

Este ejemplo busca todos los elementos secundarios del elemento `Address`.

Este ejemplo utiliza el siguiente documento XML: Archivo XML de ejemplo: Varios pedidos de compra (LINQ to XML).

```
XDocument cpo = XDocument.Load("PurchaseOrders.xml");
XElement po = cpo.Root.Element("PurchaseOrder").Element("Address");

// LINQ to XML query
IEnumerable<XElement> list1 = po.Elements();

// XPath expression
IEnumerable<XElement> list2 = po.XPathSelectElements("./*");

if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Name>Ellen Adams</Name>
<Street>123 Maple Street</Street>
<City>Mill Valley</City>
<State>CA</State>
<Zip>10999</Zip>
<Country>USA</Country>
```

Procedimiento para buscar el elemento raíz (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema muestra cómo obtener el elemento raíz con XPath y LINQ to XML.

La expresión XPath es:

`/PurchaseOrders`

Ejemplo

Este ejemplo busca el elemento raíz.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: varios pedidos de compra (LINQ to XML).

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

// LINQ to XML query
XElement el1 = po.Root;

// XPath expression
XElement el2 = po.XPathSelectElement("/PurchaseOrders");

if (el1 == el2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(el1.Name);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
PurchaseOrders
```

Procedimiento para buscar elementos descendientes (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo obtener los elementos descendientes con un nombre particular.

La expresión XPath es `//Name`.

Ejemplo

Este ejemplo busca todos los descendientes con el nombre `Name`.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: varios pedidos de compra (LINQ to XML).

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

// LINQ to XML query
IEnumerable<XElement> list1 = po.Root.Descendants("Name");

// XPath expression
IEnumerable<XElement> list2 = po.XPathSelectElements("//Name");

if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Name>Ellen Adams</Name>
<Name>Tai Yee</Name>
<Name>Cristian Osorio</Name>
<Name>Cristian Osorio</Name>
<Name>Jessica Arnold</Name>
<Name>Jessica Arnold</Name>
```

Procedimiento para filtrar por un atributo (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema muestra cómo obtener los elementos descendientes con un nombre especificado y con un atributo con un valor especificado.

La expresión XPath es:

`.//Address[@Type='Shipping']`

Ejemplo

Este ejemplo encuentra todos los elementos descendientes con el nombre `Address` y con un atributo `Type` con el valor "Shipping".

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: varios pedidos de compra (LINQ to XML).

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

// LINQ to XML query
IEnumerable<XElement> list1 =
    from el in po.Descendants("Address")
    where (string)el.Attribute("Type") == "Shipping"
    select el;

// XPath expression
IEnumerable<XElement> list2 = po.XPathSelectElements(".//Address[@Type='Shipping']");

if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Address Type="Shipping">
  <Name>Ellen Adams</Name>
  <Street>123 Maple Street</Street>
  <City>Mill Valley</City>
  <State>CA</State>
  <Zip>10999</Zip>
  <Country>USA</Country>
</Address>
<Address Type="Shipping">
  <Name>Cristian Osorio</Name>
  <Street>456 Main Street</Street>
  <City>Buffalo</City>
  <State>NY</State>
  <Zip>98112</Zip>
  <Country>USA</Country>
</Address>
<Address Type="Shipping">
  <Name>Jessica Arnold</Name>
  <Street>4055 Madison Ave</Street>
  <City>Seattle</City>
  <State>WA</State>
  <Zip>98112</Zip>
  <Country>USA</Country>
</Address>
```

Procedimiento para buscar elementos relacionados (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema muestra cómo obtener un elemento seleccionando en un atributo al que hace referencia el valor de otro elemento.

La expresión XPath es:

```
.//Customer[@CustomerID=/Root/Orders/Order[12]/CustomerID]
```

Ejemplo

Este ejemplo encuentra el duodécimo (12) elemento `Order`, y después encuentra el cliente de dicho pedido.

Tenga en cuenta que la indización en una lista de .NET se basa en "cero". Sin embargo, la indización en una colección de nodos de un predicado XPath se basa en "uno". Este ejemplo refleja esta diferencia.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

```
XDocument co = XDocument.Load("CustomersOrders.xml");

// LINQ to XML query
XElement customer1 =
    (from el in co.Descendants("Customer")
     where (string)el.Attribute("CustomerID") ==
          (string)(co
              .Element("Root")
              .Element("Orders")
              .Elements("Order")
              .ToList()[11]
              .Element("CustomerID"))
    select el)
    .First();

// An alternate way to write the query that avoids creation
// of a System.Collections.Generic.List:
XElement customer2 =
    (from el in co.Descendants("Customer")
     where (string)el.Attribute("CustomerID") ==
          (string)(co
              .Element("Root")
              .Element("Orders")
              .Elements("Order")
              .Skip(11).First()
              .Element("CustomerID"))
    select el)
    .First();

// XPath expression
XElement customer3 = co.XPathSelectElement(
  ".//Customer[@CustomerID=/Root/Orders/Order[12]/CustomerID]");

if (customer1 == customer2 && customer1 == customer3)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(customer1);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Customer CustomerID="HUNGC">
  <CompanyName>Hungry Coyote Import Store</CompanyName>
  <ContactName>Yoshi Latimer</ContactName>
  <ContactTitle>Sales Representative</ContactTitle>
  <Phone>(503) 555-6874</Phone>
  <Fax>(503) 555-2376</Fax>
  <FullAddress>
    <Address>City Center Plaza 516 Main St.</Address>
    <City>Elgin</City>
    <Region>OR</Region>
    <PostalCode>97827</PostalCode>
    <Country>USA</Country>
  </FullAddress>
</Customer>
```

Procedimiento para buscar elementos de un espacio de nombres (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Las expresiones XPath pueden buscar nodos en un espacio de nombres específico. Las expresiones XPath usan prefijos de espacio de nombres para especificar espacios de nombre. Para analizar una expresión XPath que contiene prefijos de espacio de nombre, debe pasar un objeto a los métodos XPath que implementan IXmlNamespaceResolver. Este ejemplo usa XmlNamespaceManager.

La expresión XPath es:

`./aw:*`

Ejemplo

En el ejemplo siguiente se lee un árbol XML que contiene dos espacios de nombres. Usa un objeto XmlReader para leer el documento XML. Después obtiene un objeto XmlNameTable de XmlReader y un objeto XmlNamespaceManager de XmlNameTable. Utiliza XmlNamespaceManager al seleccionar elementos.

```
XmlReader reader = XmlReader.Create("ConsolidatedPurchaseOrders.xml");
XElement root = XElement.Load(reader);
XmlNameTable nameTable = reader.NameTable;
XmlNamespaceManager namespaceManager = new XmlNamespaceManager(nameTable);
namespaceManager.AddNamespace("aw", "http://www.adventure-works.com");
IEnumerable<XElement> list1 = root.XPathSelectElements("./aw:*", namespaceManager);
IEnumerable<XElement> list2 =
    from el in root.Elements()
    where el.Name.Namespace == "http://www.adventure-works.com"
    select el;
if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list2)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<aw:PurchaseOrder PONumber="11223" Date="2000-01-15" xmlns:aw="http://www.adventure-works.com">
    <aw:ShippingAddress>
      <aw:Name>Chris Preston</aw:Name>
      <aw:Street>123 Main St.</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98113</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:ShippingAddress>
    <aw:BillingAddress>
      <aw:Name>Chris Preston</aw:Name>
      <aw:Street>123 Main St.</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98113</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:BillingAddress>
    <aw:DeliveryInstructions>Ship only complete order.</aw:DeliveryInstructions>
    <aw:Item PartNum="LIT-01">
      <aw:ProductID>Litware Networking Card</aw:ProductID>
      <aw:Qty>1</aw:Qty>
      <aw:Price>20.99</aw:Price>
    </aw:Item>
    <aw:Item PartNum="LIT-25">
      <aw:ProductID>Litware 17in LCD Monitor</aw:ProductID>
      <aw:Qty>1</aw:Qty>
      <aw:Price>199.99</aw:Price>
    </aw:Item>
  </aw:PurchaseOrder>
```

Procedimiento para buscar elementos anteriores del mismo nivel (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se compara el eje XPath `preceding-sibling` con el eje secundario de LINQ to XMLXNode.ElementsBeforeSelf.

La expresión XPath es:

`preceding-sibling::*`

Observe que los resultados de XPathSelectElements y de XNode.ElementsBeforeSelf aparecen en el mismo orden que en el documento.

Ejemplo

El siguiente ejemplo busca el elemento `FullAddress` y, a continuación, recupera los elementos anteriores utilizando el eje `preceding-sibling`.

En este ejemplo se usa el siguiente documento XML: Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML).

```
XElement co = XElement.Load("CustomersOrders.xml");

XElement add = co.Element("Customers").Element("Customer").Element("FullAddress");

// LINQ to XML query
IEnumerable<XElement> list1 = add.ElementsBeforeSelf();

// XPath expression
IEnumerable<XElement> list2 = add.XPathSelectElements("preceding-sibling::*");

if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list2)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<CompanyName>Great Lakes Food Market</CompanyName>
<ContactName>Howard Snyder</ContactName>
<ContactTitle>Marketing Manager</ContactTitle>
<Phone>(503) 555-7555</Phone>
```

Procedimiento para buscar descendientes de un elemento secundario (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo obtener los elementos descendientes de un elemento secundario con un nombre particular.

La expresión XPath es:

```
./Paragraph//Text/text()
```

Ejemplo

Este ejemplo simula los problemas de extraer texto de una representación XML de un documento de un procesador de texto. Primero selecciona todos los elementos de `Paragraph` y después selecciona todos los elementos descendientes de `Text` de cada elemento `Paragraph`. Esto no selecciona los elementos `Text` descendientes del elemento `Comment`.

```
XElement root = XElement.Parse(
@"<Root>
  <Paragraph>
    <Text>This is the start of</Text>
  </Paragraph>
  <Comment>
    <Text>This comment is not part of the paragraph text.</Text>
  </Comment>
  <Paragraph>
    <Annotation Emphasis='true'>
      <Text> a sentence.</Text>
    </Annotation>
  </Paragraph>
  <Paragraph>
    <Text>  This is a second sentence.</Text>
  </Paragraph>
</Root>");

// LINQ to XML query
string str1 =
    root
    .Elements("Paragraph")
    .Descendants("Text")
    .Select(s => s.Value)
    .Aggregate(
        new StringBuilder(),
        (s, i) => s.Append(i),
        s => s.ToString()
    );

// XPath expression
string str2 =
    ((IEnumerable)root.XPathEvaluate("./Paragraph//Text/text()"))
    .Cast<XText>()
    .Select(s => s.Value)
    .Aggregate(
        new StringBuilder(),
        (s, i) => s.Append(i),
        s => s.ToString()
    );

if (str1 == str2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(str2);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
This is the start of a sentence.  This is a second sentence.
```

Procedimiento para buscar la unión de dos rutas de acceso a ubicaciones (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

XPath permite buscar la unión de los resultados de dos rutas de ubicación XPath.

La expresión XPath es:

`//Category|//Price`

Puede conseguir los mismos resultados usando el operador de consulta estándar Concat.

Ejemplo

Este ejemplo busca todos los elementos `Category` y todos los elementos `Price` y los concatena en una única recopilación. Tenga en cuenta que la consulta LINQ to XML llama a InDocumentOrder para ordenar los resultados. Los resultados de la evaluación de la expresión XPath también están en el orden del documento.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Datos numéricos (LINQ to XML).

```
XDocument data = XDocument.Load("Data.xml");

// LINQ to XML query
IEnumerable<XElement> list1 =
    data
    .Descendants("Category")
    .Concat(
        data
        .Descendants("Price")
    )
    .InDocumentOrder();

// XPath expression
IEnumerable<XElement> list2 = data.XPathSelectElements("//Category|//Price");

if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Category>A</Category>
<Price>24.50</Price>
<Category>B</Category>
<Price>89.99</Price>
<Category>A</Category>
<Price>4.95</Price>
<Category>A</Category>
<Price>66.00</Price>
<Category>B</Category>
<Price>.99</Price>
<Category>A</Category>
<Price>29.00</Price>
<Category>B</Category>
<Price>6.99</Price>
```

Procedimiento para buscar nodos del mismo nivel (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Quizás desea buscar todos los elementos relacionados de un nodo que tienen un nombre específico. La recopilación resultante puede incluir el nodo de contexto si también tiene el nombre específico.

La expresión XPath es:

`../Book`

Ejemplo

Este ejemplo primero busca un elemento `Book` y después busca todos los elementos secundarios con el nombre `Book`. La colección resultante incluye el nodo de contexto.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: libros (LINQ to XML).

```
XDocument books = XDocument.Load("Books.xml");

XElement book =
    books
    .Root
    .Elements("Book")
    .Skip(1)
    .First();

// LINQ to XML query
IEnumerable<XElement> list1 =
    book
    .Parent
    .Elements("Book");

// XPath expression
IEnumerable<XElement> list2 = book.XPathSelectElements("../Book");

if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Book id="bk101">
  <Author>Garghentini, Davide</Author>
  <Title>XML Developer's Guide</Title>
  <Genre>Computer</Genre>
  <Price>44.95</Price>
  <PublishDate>2000-10-01</PublishDate>
  <Description>An in-depth look at creating applications
      with XML.</Description>
</Book>
<Book id="bk102">
  <Author>Garcia, Debra</Author>
  <Title>Midnight Rain</Title>
  <Genre>Fantasy</Genre>
  <Price>5.95</Price>
  <PublishDate>2000-12-16</PublishDate>
  <Description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</Description>
</Book>
```

Procedimiento para buscar un atributo del elemento principal (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo desplazarse hasta el elemento primario y buscar un atributo en él.

La expresión XPath es:

`../@id`

Ejemplo

Este ejemplo busca primero un elemento `Author`. Después busca el atributo `id` del elemento primario.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: libros (LINQ to XML).

```
XDocument books = XDocument.Load("Books.xml");

XElement author =
    books
    .Root
    .Element("Book")
    .Element("Author");

// LINQ to XML query
XAttribute att1 =
    author
    .Parent
    .Attribute("id");

// XPath expression
XAttribute att2 = ((IEnumerable)author.XPathEvaluate("../@id")).Cast<XAttribute>().First();

if (att1 == att2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(att1);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
id="bk101"
```

Procedimiento para buscar atributos de elementos del mismo nivel con un nombre específico (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo buscar todos los atributos de los elementos secundarios del nodo de contexto. Sólo se devuelven los atributos con un nombre específico en la recopilación.

La expresión XPath es:

`../Book/@id`

Ejemplo

Este ejemplo primero busca un elemento `Book`, después busca todos los elementos secundarios con el nombre `Book` y después busca todos los atributos con el nombre `id`. El resultado es una colección de atributos.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: libros (LINQ to XML).

```
XDocument books = XDocument.Load("Books.xml");

XElement book =
    books
    .Root
    .Element("Book");

// LINQ to XML query
IEnumerable<XAttribute> list1 =
    from el in book.Parent.Elements("Book")
    select el.Attribute("id");

// XPath expression
IEnumerable<XAttribute> list2 =
  ((IEnumerable)book.XPathEvaluate("../Book/@id")).Cast<XAttribute>();

if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XAttribute el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
id="bk101"
id="bk102"
```

Procedimiento para buscar elementos con un atributo específico (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En ocasiones, deseará buscar todos los elementos que tengan un atributo en particular. Pero no le preocupa cuáles es el contenido del atributo. En vez de ello, desea realizar la selección en función de si existe o no el atributo.

La expresión XPath es:

`./*[@Select]`

Ejemplo

El siguiente código selecciona únicamente los elementos que tengan el atributo `Select`.

```
XElement doc = XElement.Parse(
@"<Root>
    <Child1>1</Child1>
    <Child2 Select='true'>2</Child2>
    <Child3>3</Child3>
    <Child4 Select='true'>4</Child4>
    <Child5>5</Child5>
</Root>");

// LINQ to XML query
IEnumerable<XElement> list1 =
    from el in doc.Elements()
    where el.Attribute("Select") != null
    select el;

// XPath expression
IEnumerable<XElement> list2 =
    ((IEnumerable)doc.XPathEvaluate("./*[@Select]")).Cast<XElement>();

if (list1.Count() == list2.Count() &&
        list1.Intersect(list2).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Child2 Select="true">2</Child2>
<Child4 Select="true">4</Child4>
```

Procedimiento para buscar elementos secundarios en función de la posición (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

En ocasiones, deseará buscar elementos en función de su posición. Quizá desee buscar el segundo elemento o buscar el tercero en el quinto elemento.

La expresión XPath es:

`Test[position() >= 2 and position() <= 4]`

Existen dos aproximaciones posibles para escribir esta consulta de LINQ to XML de forma diferida. Puede utilizar los operadores Skip y Take, o bien utilizar la sobrecarga Where que recibe un índice. Si utiliza la sobrecarga Where, estará utilizando una expresión lambda que recibe dos argumentos. El siguiente ejemplo muestra ambos métodos para seleccionar elementos en base a la posición.

Ejemplo

Este ejemplo encontrará el segundo en el cuarto elemento de `Test`. El resultado es una colección de elementos.

En este ejemplo se usa el siguiente documento XML: Archivo XML de muestra: Configuración de prueba (LINQ to XML).

```
XElement testCfg = XElement.Load("TestConfig.xml");

// LINQ to XML query
IEnumerable<XElement> list1 =
    testCfg
    .Elements("Test")
    .Skip(1)
    .Take(3);

// LINQ to XML query
IEnumerable<XElement> list2 =
    testCfg
    .Elements("Test")
    .Where((el, idx) => idx >= 1 && idx <= 3);

// XPath expression
IEnumerable<XElement> list3 =
  testCfg.XPathSelectElements("Test[position() >= 2 and position() <= 4]");

if (list1.Count() == list2.Count() &&
    list1.Count() == list3.Count() &&
    list1.Intersect(list2).Count() == list1.Count() &&
    list1.Intersect(list3).Count() == list1.Count())
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
foreach (XElement el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Test TestId="0002" TestType="CMD">
  <Name>Find succeeding characters</Name>
  <CommandLine>Examp2.EXE</CommandLine>
  <Input>abc</Input>
  <Output>def</Output>
</Test>
<Test TestId="0003" TestType="GUI">
  <Name>Convert multiple numbers to strings</Name>
  <CommandLine>Examp2.EXE /Verbose</CommandLine>
  <Input>123</Input>
  <Output>One Two Three</Output>
</Test>
<Test TestId="0004" TestType="GUI">
  <Name>Find correlated key</Name>
  <CommandLine>Examp3.EXE</CommandLine>
  <Input>a1</Input>
  <Output>b1</Output>
</Test>
```

Procedimiento para buscar el nodo del mismo nivel inmediatamente anterior (XPath-LINQ to XML) (C#)

18/03/2020 • 2 minutes to read • Edit Online

Es posible que en alguna ocasión desee buscar el elemento del mismo nivel inmediatamente anterior a un nodo. Debido a las diferencias de semántica de los predicados posicionales para los ejes de los elementos del mismo nivel anteriores de XPath en comparación con LINQ to XML, ésta es una de las comparaciones más interesantes.

Ejemplo

En este ejemplo, la consulta LINQ to XML usa el operador Last para buscar el último nodo de la recopilación devuelto por ElementsBeforeSelf. Por el contrario, la expresión XPath usa un predicado con un valor de 1 para buscar el elemento inmediatamente anterior.

```
XElement root = XElement.Parse(
    @"<Root>
    <Child1/>
    <Child2/>
    <Child3/>
    <Child4/>
</Root>");
XElement child4 = root.Element("Child4");

// LINQ to XML query
XElement el1 =
    child4
    .ElementsBeforeSelf()
    .Last();

// XPath expression
XElement el2 =
    ((IEnumerable)child4
                  .XPathEvaluate("preceding-sibling::*[1]"))
                  .Cast<XElement>()
                  .First();

if (el1 == el2)
    Console.WriteLine("Results are identical");
else
    Console.WriteLine("Results differ");
Console.WriteLine(el1);
```

Este ejemplo produce el siguiente resultado:

```
Results are identical
<Child3 />
```

Introducción a las transformaciones funcionales puras (C#)

18/03/2020 • 2 minutes to read • Edit Online

Esta sección ofrece una introducción a las transformaciones funcionales, incluyendo los conceptos subyacentes y los constructores del lenguaje que las hacen posible. Realiza una comparación entre las metodologías de programación de orientación a objetos y de transformación funcional, incluyendo consejos acerca de cómo llevar a cabo la transición a ésta última. Aunque es posible utiliza las transformaciones funcionales en numerosos escenarios, aquí utilizaremos la transformación XML como un ejemplo concreto.

En Tutorial: Manipulación de contenido en un documento WordprocessingML (C#) se proporcionan una serie de ejemplos, cada uno basado en el anterior. Esos ejemplos demuestran el enfoque de transformación funcional pura para manipular XML. Este tutorial presupone un conocimiento práctico de C#. En este tutorial no se proporciona la semántica detallada de las construcciones del lenguaje, pero se proporcionan vínculos a documentación del lenguaje según corresponda.

También se asume un conocimiento práctico de los conceptos básicos de informática básica y XML, incluyendo espacios de nombres XML.

En esta sección

TEMA	DESCRIPCIÓN
Conceptos y terminología (transformación funcional) (C#)	Presenta los conceptos y la terminología en relación con las transformaciones funcionales.
Diferencias entre la programación funcional y programación imperativa (C#)	Compara y contrasta la programación funcional con la programación imperativa (orientada a procedimientos) más tradicional.
Refactorizar en funciones puras (C#)	Presenta las funciones puras y muestra ejemplos de funciones puras e impuras.
Aplicabilidad de la transformación funcional (C#)	Describe los escenarios típicos donde se realizan transformaciones funcionales.
Transformación funcional de XML (Visual Basic)	Describe las transformaciones funcionales en el contexto de los árboles XML de transformación.

Conceptos y terminología (transformación funcional) (C#)

18/03/2020 • 8 minutes to read • Edit Online

Este tema presenta los conceptos y la terminología en relación con las transformaciones funcionales. La aproximación que utiliza la transformación funcional para transformar datos permite obtener un código que normalmente se escribe más rápido, es más expresivo y fácil de mantener y depurar, si lo comparamos con la programación imperativa, que es más tradicional.

Observe que los temas incluidos en esta lección no pretenden explicar con detalle cómo es la programación funcional. En su lugar, pretenden resaltar algunas de las características de la programación funcional que facilitan el proceso de transformar un XML de una forma a otra.

¿En qué consiste la transformación funcional pura?

En la *transformación funcional pura*, existe un conjunto de funciones, llamadas *funciones puras*, que definen cómo transformar un conjunto de datos estructurados de su forma original a otra forma. La palabra "pura" indica que las funciones *admiten composición*, lo que significa que son:

- *Autocontenidas*, de forma que se pueden ordenar o reorganizar libremente sin preocuparse por las dependencias que puedan tener con el resto del programa. Las transformaciones puras no tienen conocimiento sobre su entorno ni ningún efecto sobre éste. Es decir, las funciones que se usan en la transformación no tienen *efectos secundarios*.
- *Sin estado*, lo que significa que si se ejecuta la misma función o un conjunto específico de funciones en los mismos datos de entrada, el resultado siempre será el mismo. Las transformaciones puras no recuerdan sus ejecuciones anteriores.

> **IMPORTANT**
>
> Para el resto de este tutorial, el término "función pura" se utilizará en sentido general para señalar una técnica de programación y no una característica específica del lenguaje.
>
> Tenga en cuenta que las funciones puras deben implementarse como métodos en C#.
>
> Además, no debería confundir las funciones puras con los métodos virtuales puros de C++. Estos últimos indican que la clase contenedora es abstracta y que no incluye ningún cuerpo de método.

Programación funcional

La *programación funcional* es una técnica de programación que permite el uso de transformaciones funcionales puras directamente.

Tradicionalmente, los lenguajes de programación funcionales de uso general, como ML, Scheme, Haskell y F#, han atraído principalmente la atención de la comunidad académica. Aunque siempre ha sido posible escribir transformaciones funcionales puras en C#, la dificultad para hacerlo no la ha convertido en una opción atractiva de cara a los programadores. No obstante, en las versiones recientes de C# se han incorporado nuevos constructores del lenguaje, como son las expresiones lambda o la inferencia de tipos, que simplifican enormemente la programación funcional y permiten aumentar la productividad.

Para obtener más información sobre la programación funcional, consulte Diferencias entre la programación funcional y la programación imperativa (C#).

Lenguajes de programación funcionales para ciertos campos

Aunque todavía no se han adoptado mayoritariamente los lenguajes de programación funcionales, sí han tenido más éxito los lenguajes de programación funcionales específicos para ciertos campos. Por ejemplo, las hojas de estilos en cascada (CSS) se utilizan para definir la apariencia de numerosas paginas web y las hojas de estilos del Lenguaje de transformación basado en hojas de estilo (XSLT) se utilizan muy a menudo para la manipulación de datos XML. Para obtener más información sobre XSLT, consulte Transformaciones XSLT.

Terminología

La siguiente tabla define algunos términos relacionados con las transformaciones funcionales.

Función de orden superior (primera clase)
Función que se puede tratar como un objeto de programación. Por ejemplo, es posible pasar una función de orden superior como argumento de otra función, así como ser devuelta por otra función. En C#, los delegados y las expresiones lambda son características del lenguaje que admiten el uso de funciones de orden superior. Si desea escribir una función de orden superior, deberá declarar uno o más argumentos para recibir delegados, y a menudo utilizará expresiones lambda cuando llame a dichas funciones. La mayoría de operadores estándar de consulta son funciones de orden superior.

Para obtener más información, vea Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#)).

Expresión lambda
En esencia, es posible utilizar una función anónima alineada siempre que se espere como argumento un tipo delegado. Esta es una definición simplificada de las expresiones lambda, pero resulta adecuada para los objetivos de este tutorial.

Para obtener más información, consulte Expresiones lambda.

Colección
Conjunto de datos estructurados, normalmente del mismo tipo. Para que una colección sea compatible con LINQ, ésta debe implementar la interfaz IEnumerable o la interfaz IQueryable (o una de sus equivalentes genéricas), IEnumerator<T> o IQueryable<T>).

Tupla (tipos anónimos)
Se trata de un concepto matemático: una tupa es una secuencia finita de objetos, cada uno de los cuales es de un tipo específico. A las tuplas también se las conoce como listas ordenadas. Los tipos anónimos son una implementación del lenguaje para este concepto, lo que permite declarar un tipo de clase sin nombre e instanciar un objeto de este tipo al mismo tiempo.

Para obtener más información, consulte Tipos anónimos (Guía de programación de C#).

Inferencia de tipos (tipos implícitos)
Consiste en la capacidad del compilador para determinar el tipo de una variable en caso de que no exista una declaración de tipos explícita.

Para más información, vea Variables locales con asignación implícita de tipos.

Ejecución aplazada y evaluación diferida
Es posible retrasar la evaluación de una expresión hasta que se requiera el valor resultante. Las colecciones admiten la ejecución aplazada.

Para obtener más información, consulte Introducción a las consultas LINQ (C#) y Ejecución aplazada y evaluación diferida en LINQ to XML.

Estas características del lenguaje se utilizarán en códigos de ejemplo a lo largo de esta sección.

Vea también

- Introducción a las transformaciones funcionales puras (C#)
- Diferencias entre la programación funcional y la programación imperativa (C#)

Diferencias entre la programación funcional y La programación imperativa (C#)

18/03/2020 • 8 minutes to read • Edit Online

Este tema compara y contrasta la programación funcional y la programación imperativa (orientada a procedimientos) más tradicional.

Diferencias entre la programación funcional y Programación imperativa

El paradigma de la *programación funcional* se ha creado explícitamente para permitir un enfoque puramente funcional de la resolución de problemas. La programación funcional es una forma de *programación declarativa*. Por el contrario, la mayoría de lenguajes más populares, incluidos los lenguajes de programación orientada a objetos (OOP) como C#, Visual Basic, C++ y Java, se han diseñado para admitir en primer lugar la programación *imperativa* (orientada a procedimientos).

El enfoque imperativo permite al desarrollador escribir código que describe detalladamente los pasos que el equipo debe realizar para cumplir el objetivo. A veces también se denomina programación *algorítmica*. Por el contrario, un enfoque funcional implica crear el problema como un conjunto de funciones que se deben ejecutar. Es necesario definir con cuidado la entrada a cada función y qué devuelve cada función. La siguiente tabla describe algunas de las diferencias generales entre estos dos enfoques.

CARACTERÍSTICA	ENFOQUE IMPERATIVO	ENFOQUE FUNCIONAL
Enfoque del programador	Cómo realizar tareas (algoritmos) y cómo realizar el seguimiento de cambios de estado.	Información deseada y transformaciones necesarias
Cambios de estado	Importante	Inexistente
Orden de ejecución	Importante	Baja importancia
Control del flujo primario	Bucles, elementos condicionales y llamadas a funciones (métodos).	Llamadas a funciones, incluyendo la recursividad.
Unidad de manipulación primaria	Instancias de estructuras o clases	Funciones como recopilaciones de datos y objetos de primera clase

Aunque la mayoría de lenguajes se diseñaron para admitir un paradigma de programación específico, muchos lenguajes generales son lo suficientemente flexibles para admitir varios paradigmas. Por ejemplo, la mayoría de los lenguajes que contienen punteros a funciones se pueden usar para permitir de forma creíble la programación funcional. Además, C# incluye extensiones de lenguaje explícitas para que sean compatibles con la programación funcional, incluidas las expresiones lambda e inferencia de tipos. La tecnología LINQ es una forma de programación funcional y declarativa.

Programación funcional con XSLT

Muchos desarrolladores de XSLT están familiarizados con el enfoque funcional puro. La forma más eficaz de desarrollar una hoja de estilos XSLT es tratar cada plantilla como una transformación de componentes aislada. El orden de ejecución no está realzado. XSLT no admite efectos secundarios (excepto cuando mecanismos de escape para ejecutar código de procedimiento puedan provocar efectos secundarios, cuyo resultado es una impureza

funcional). Sin embargo, a pesar de que XSLT es una herramienta eficaz, algunas de sus características no son óptimas. Por ejemplo, cuando se expresan construcciones de programación en XML se obtiene un código relativamente detallado y, por lo tanto, difícil de mantener. Asimismo, la gran dependencia de la repetición para el control de flujo puede provocar que el código sea difícil de leer. Para obtener más información sobre XSLT, vea Transformaciones XSLT.

No obstante, XSLT ha demostrado el valor de usar un enfoque funcional puro para transformar XML de una forma a otra. La programación funcional pura con LINQ to XML es muy parecida a XSLT. En cambio, las construcciones de programación introducidas por LINQ to XML y C# le permiten escribir transformaciones funcionales puras que son más fáciles de leer y mantener que XSLT.

Ventajas de las funciones puras

El motivo principal para implementar transformaciones funcionales como funciones puras es que las funciones puras son ajustables: es decir, independientes y sin estado. Estas características aportan varias ventajas, incluyendo las siguientes:

- Mayor legibilidad y facilidad de mantenimiento. Esto se debe a que cada función está diseñada para cumplir una tarea específica dependiendo de sus argumentos. Esta función no depende de ningún estado externo.
- Desarrollo reiterativo más sencillo. Como es más sencillo refactorizar el código, la implementación de los cambios de diseño resulta a menudo más fácil. Por ejemplo, supongamos que escribe una transformación complicada y después se da cuenta de que parte del código se repite varias veces en la transformación. Si refactoriza mediante un método puro, puede llamar a su método puro cuando lo desee sin preocuparse de efectos secundarios.
- Pruebas y depuraciones más sencillas. Como las funciones puras se pueden probar más fácilmente en aislamiento, puede escribir código de prueba que llame a la función pura con valores típicos, casos avanzados válidos y casos avanzados no válidos.

Transición para desarrolladores de OOP

En la programación orientada a objetos (OOP) tradicional, la mayoría de desarrolladores están acostumbrados a programar en estilo imperativo/de procedimientos. Para pasar al desarrollo en un estilo funcional puro, tienen que realizar una transición en su forma de pensar y en su planteamiento del desarrollo.

Para solucionar problemas, los desarrolladores de OOP diseñan jerarquías de clases, se centran en la correcta encapsulación y piensan en términos de contratos de clase. El comportamiento y el estado de los tipos de objetos son de vital importancia y las características del lenguaje, como las clases, las interfaces, la herencia y el polimorfismo se proporcionan para tratar estas preocupaciones.

Por el contrario, la programación funcional considera los problemas de cálculo como un ejercicio de evaluación de transformaciones funcionales puras de recopilaciones de datos. La programación funcional evita datos mutables y de estado, y pone de relieve, en su lugar, la aplicación de las funciones.

Afortunadamente, C# no requiere un salto completo a la programación funcional, ya que admite enfoques de programación imperativos y funcionales. Un desarrollador puede elegir el enfoque más adecuado para una escenario específico. En realidad, los programas suelen combinar ambos enfoques.

Vea también

- Introducción a las transformaciones funcionales puras (C#)
- Transformaciones XSLT
- Refactorizar en funciones puras (C#)

Refactorizar en funciones puras (C#)

18/03/2020 • 6 minutes to read • Edit Online

Un aspecto importante de las transformaciones funcionales puras es aprender cómo refactorizar código usando funciones puras.

> **NOTE**
> La nomenclatura común en la programación funcional consiste en refactorizar programas usando funciones puras. En Visual Basic y C++, esto se alinea con el uso de funciones en los lenguajes respectivos. No obstante, en C#, las funciones son métodos llamados. Con fines de análisis, una función pura se implementa como un método en C#.

Tal como se indicó previamente en esta sección, una función pura tiene dos características útiles:

- No tiene efectos secundarios. La función no cambia variables o datos de ningún tipo fuera de la función.
- Es coherente. Con el mismo conjunto de datos de entrada, siempre devolverá el mismo valor de salida.

Una forma de realizar una transición a la programación funcional es refactorizar código existente para eliminar efectos secundarios innecesarios y dependencias externas. De esta forma puede crear versiones de función pura del código existente.

En este tema se trata qué es una función pura y qué no es. En Tutorial: Manipulación de contenido en un documento WordprocessingML (C#) se muestra cómo manipular un documento de WordprocessingML y se incluyen dos ejemplos de cómo refactorizar mediante una función pura.

Eliminar efectos secundarios y dependencias externas

Los siguientes ejemplos contraponen dos funciones no puras y una función pura.

Función no pura que cambia un miembro de clase

En el siguiente código, la función `HyphenatedConcat` no es una función pura porque modifica el miembro de datos `aMember` en la clase:

```
public class Program
{
    private static string aMember = "StringOne";

    public static void HyphenatedConcat(string appendStr)
    {
        aMember += '-' + appendStr;
    }

    public static void Main()
    {
        HyphenatedConcat("StringTwo");
        Console.WriteLine(aMember);
    }
}
```

Este código genera el siguiente resultado:

```
StringOne-StringTwo
```

Tenga en cuenta que es irrelevante si los datos que se están modificando tienen acceso a `public` o `private`, o si son un miembro de `static` o un miembro de instancia. Una función pura no cambia datos fuera de la función.

Función no pura que cambia un argumento

Asimismo, la siguiente versión de esta misma función no es pura porque modifica el contenido de su parámetro, `sb`.

```
public class Program
{
    public static void HyphenatedConcat(StringBuilder sb, String appendStr)
    {
        sb.Append('-' + appendStr);
    }

    public static void Main()
    {
        StringBuilder sb1 = new StringBuilder("StringOne");
        HyphenatedConcat(sb1, "StringTwo");
        Console.WriteLine(sb1);
    }
}
```

Esta versión del programa crea el mismo resultado que la primera versión porque la función `HyphenatedConcat` ha cambiado el valor (estado) de su primer parámetro invocando la función de miembro Append. Tenga en cuenta que esta modificación se produce a pesar del hecho que `HyphenatedConcat` usar el paso de parámetros llamada por valor.

> **IMPORTANT**
> Para los tipos de referencia, si pasa un parámetro por valor, tiene como resultado una copia de la referencia a un objeto que se está pasando. Esta copia sigue asociada con los mismos datos de la instancia que la referencia original (hasta que la variable de referencia se asigna a un objeto nuevo). La llamada por referencia no es necesariamente requerida para que una función modifique un parámetro.

Función pura

La versión siguiente del programa muestra cómo implementar la función `HyphenatedConcat` como una función pura.

```
class Program
{
    public static string HyphenatedConcat(string s, string appendStr)
    {
        return (s + '-' + appendStr);
    }

    public static void Main(string[] args)
    {
        string s1 = "StringOne";
        string s2 = HyphenatedConcat(s1, "StringTwo");
        Console.WriteLine(s2);
    }
}
```

De nuevo, esta versión crea la misma línea de salida: `StringOne-StringTwo`. Tenga en cuenta que para conservar un

valor concatenado, se almacena en la variable intermedia `s2`.

Un enfoque que puede ser muy útil para escribir funciones que son localmente impuras (es decir, declaran y modifican variables locales) pero que son globalmente puras. Tales funciones tienen muchas características deseables de facilidad de creación, pero evitan algunas de las expresiones de programación funcional más complicadas, como tener que usar una recursión cuando un simple bucle lograría lo mismo.

Operadores de consulta estándar

Una característica importante de los operadores de consulta estándar es que se implementan como funciones puras.

Para obtener más información, vea Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#)).

Vea también

- Introducción a las transformaciones funcionales puras (C#)
- Diferencias entre la programación funcional y la programación imperativa (C#)

Aplicabilidad de la transformación funcional (C#)

18/03/2020 • 4 minutes to read • Edit Online

Las transformaciones funcionales puras se pueden aplicar en una amplia variedad de situaciones.

El enfoque de transformación funcional es especialmente adecuado para consultar y manipular datos estructurados; por lo tanto encaja bien con las tecnologías LINQ. No obstante, la transformación funcional tiene una aplicabilidad mucho más amplia que el uso con LINQ. Cualquier proceso en el que el centro de atención principal sea la transformación de datos de un formato a otro probablemente debe considerarse un candidato para la transformación funcional.

Este enfoque es aplicable a muchos problemas que a primera vista pueden no parecer candidatos. La transformación funcional, usada en conjunción con o de forma separada de LINQ, debe tenerse en cuenta para las siguientes áreas:

- Documentos basados en XML. Los datos bien formados de cualquier dialecto XML se pueden manipular fácilmente mediante la transformación funcional. Para obtener más información, consulte Transformación funcional de XML (C#).
- Otros formatos de archivo estructurados. Desde archivos Windows.ini a documentos de texto sin formato, la mayoría de archivos tienen alguna estructura que se presta a análisis y transformación.
- Protocolos de secuencia de datos. La codificación y decodificación de datos de protocolos de comunicación a menudo se puede representar con una sencilla transformación funcional.
- Datos RDBMS y OODBMS. Las bases de datos relacionales y orientadas a objetos, como XML, son orígenes de datos estructurados muy utilizados.
- Soluciones científicas, estadísticas y matemáticas. Estos campos suelen manipular grandes cantidades de datos para ayudar al usuario a visualizar, estimar o resolver problemas que no son triviales.

Como se describe en Refactorizar en funciones puras (C#), el uso de funciones puras es un ejemplo de programación funcional. Además de sus ventajas inmediatas, el uso de funciones puras proporciona una valiosa experiencia para la resolución de problemas desde una perspectiva de transformación funcional. Este enfoque también puede tener un gran impacto en el diseño de clases y la programación. Éste es el caso cuando un problema se presta a una solución de transformación de datos tal como se ha escrito anteriormente.

Aunque quedan fuera del ámbito de este tutorial, los diseños que tienen la influencia de la perspectiva de transformación funcional tienden a centrarse en procesos más que en objetos como actores y la solución resultante tiende a implementarse como una serie de transformaciones a gran escala en lugar de cambios de estado de objeto individuales.

Una vez más, no olvide que C# admite los enfoques imperativo y funcional, por lo que el mejor diseño para su aplicación podría incorporar elementos de ambos.

Vea también

- Introducción a las transformaciones funcionales puras (C#)
- Transformación funcional de XML (C#)
- Refactorizar en funciones puras (C#)

Transformación funcional de XML (C#)

18/03/2020 • 4 minutes to read • Edit Online

Este tema trata acerca de la aproximación enfocada a la transformación funcional pura para modificar documentos XML, comparándola con la aproximación basada en procedimientos.

Modificar un documento XML

Una de las tareas más comunes que deben llevar a cabo los programadores de XML es la de transformar XML de una forma a otra. La forma de un documento XML es la estructura del documento, que incluye lo siguiente:

- La jerarquía expresada por el documento.
- Los nombres de elementos y atributos.
- Los tipos de datos de los elementos y atributos.

En general, la aproximación que resulta más efectiva a la hora de transformar un XML de una forma a otra, es realizar una transformación funcional pura. De esta forma, la principal tarea del programador es crear una transformación que se aplique a todo el documento XML (o a uno o más nodos estrictamente definidos). Se puede argüir que la transformación funcional es la más sencilla de programar (siempre y cuando el programador esté familiarizado con esta técnica), que genera un código muy fácil de mantener y que, a menudo, es una alternativa más compacta que las otras.

Tecnologías de transformación funcional de XML

Microsoft ofrece dos tecnologías de transformación funcional que se pueden usar con documentos XML: XSLT y LINQ to XML. XSLT se proporciona a través del espacio de nombres administrado System.Xml.Xsl y de la implementación nativa COM de MSXML. Aunque XSLT es una tecnología robusta para manipular documentos XML, requiere experiencia en un campo muy especializado, el lenguaje XSLT y sus API de apoyo.

LINQ to XML proporciona las herramientas necesarias para programar potentes transformaciones funcionales puras mediante expresiones, utilizando código C# o Visual Basic. Por ejemplo, muchos de los ejemplos incluidos en la documentación de LINQ to XML utilizan una aproximación funcional pura. Además, en el Tutorial: Manipulación de contenido en un documento WordprocessingML (C#), se usa LINQ to XML con un enfoque funcional para manipular la información de un documento de Microsoft Word.

Para obtener una comparación más completa entre LINQ to XML y otras tecnologías XML de Microsoft, vea LINQ to XML frente a otras tecnologías XML.

XSLT es una herramienta recomendada para transformaciones basadas en el documento, cuando el origen de éste tiene una estructura irregular. No obstante, LINQ to XML también puede realizar transformaciones basadas en el documento. Para más información, consulte Procedimiento para usar anotaciones para transformar árboles LINQ to XML en un estilo XSLT (C#).

Vea también

- Introducción a las transformaciones funcionales puras (C#)
- Tutorial: Manipulación de contenido en un documento WordprocessingML (C#)
- LINQ to XML frente a otras tecnologías XML

Ejecución aplazada y evaluación diferida en LINQ to XML (C#)

18/03/2020 • 3 minutes to read • Edit Online

Las operaciones de consulta y de eje a menudo se implementan para usar la ejecución aplazada. Este tema describe los requisitos y las ventajas de la ejecución aplazada, y algunas consideraciones acerca de la implementación.

Ejecución aplazada

La ejecución aplazada significa que la evaluación de una expresión se retrasa hasta que su valor *realizado* sea realmente necesario. La ejecución aplazada puede mejorar considerablemente el rendimiento cuando deba manipular grandes recolecciones de datos, sobre todo en programas que contengan una serie de manipulaciones o consultas encadenadas. En el mejor de los casos, la ejecución aplazada solo habilita una iteración a través de la recolección de origen.

Las tecnologías LINQ usan intensivamente la ejecución aplazada tanto en los miembros de las clases System.Linq principales como en los métodos de extensión de los diversos espacios de nombres LINQ, por ejemplo, System.Xml.Linq.Extensions.

El lenguaje C# admite directamente la ejecución aplazada mediante la palabra clave yield (en forma de instrucción `yield-return`) cuando se usa en un bloque de iteradores. Este tipo de iterador debe devolver una colección de tipo IEnumerator o IEnumerator<T> (o bien un tipo derivado).

Diferencias entre la evaluación diligente y la evaluación diferida

Al escribir un método que implemente una ejecución aplazada, también debe decidir si implementa el método con la evaluación diferida o la evaluación diligente.

- En la *evaluación diferida*, un único elemento de la colección de origen se procesa durante cada llamada al iterador. Esta es la forma habitual en la que se implementan los iteradores.
- En la *evaluación diligente*, la primera llamada al iterador hará que se procese toda la colección. Es posible que también se requiera una copia temporal de la colección de origen. Por ejemplo, el método OrderBy debe ordenar toda la colección antes de devolver el primer elemento.

Normalmente, la evaluación diferida presenta un mejor rendimiento, ya que distribuye el procesamiento de sobrecarga de forma equitativa durante la evaluación de la colección y minimiza el uso de los datos temporales. Obviamente, para algunas operaciones, no existe otra alternativa que materializar resultados intermedios.

Pasos siguientes

El siguiente tema de este tutorial ilustra la ejecución aplazada:

- Ejemplo de ejecución diferida (C#)

Vea también

- Tutorial: encadenar cadenas juntas (C#)
- Conceptos y terminología (transformación funcional) (C#)
- Aggregation Operations (C#) (Operaciones de agregación [C#])

- yield

Ejemplo de ejecución diferida (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo la ejecución aplazada y la evaluación diferid afectan a la ejecución de las consultas de LINQ to XML.

Ejemplo

El siguiente ejemplo muestra el orden de ejecución cuando se usa un método de extensión que utiliza la ejecución aplazada. El ejemplo declara una matriz de tres cadenas. A continuación recorre en iteración la recopilación devuelta por `ConvertCollectionToUpperCase`.

```
public static class LocalExtensions
{
    public static IEnumerable<string>
      ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source {0}", str);
            yield return str.ToUpper();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        var q = from str in stringArray.ConvertCollectionToUpperCase()
                select str;

        foreach (string str in q)
            Console.WriteLine("Main: str {0}", str);
    }
}
```

Este ejemplo produce el siguiente resultado:

```
ToUpper: source abc
Main: str ABC
ToUpper: source def
Main: str DEF
ToUpper: source ghi
Main: str GHI
```

Tenga en cuenta que durante el recorrido en iteración de la recopilación devuelta por `ConvertCollectionToUpperCase`, cada elemento se recupera de la matriz de cadenas de origen y se convierte a mayúsculas antes de que se recupere el siguiente elemento de la matriz de cadenas de origen.

Puede ver que no se convierte a mayúsculas toda la matriz de cadenas antes de que se procese cada elemento en el bucle `foreach` de `Main`.

En el siguiente tema de este tutorial se ilustra el encadenado de cadenas juntas:

- Chaining Queries Example (C#) (Ejemplo de encadenamiento de consultas (C#))

Vea también

- Tutorial: encadenar cadenas juntas (C#)

Ejemplo de encadenamiento de consultas (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este ejemplo se basa en el ejemplo anterior y muestra qué sucede cuando se encadenan dos consultas que usan ejecución aplazada y evaluación diferida.

Ejemplo

En este ejemplo se presenta otro método de extensión, `AppendString`, que anexa una cadena especificada a cada cadena de la recopilación de origen y después crea las nuevas cadenas.

```
public static class LocalExtensions
{
    public static IEnumerable<string>
      ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
      AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        foreach (string str in source)
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}
```

Este ejemplo produce el siguiente resultado:

```
ToUpper: source >abc<
AppendString: source >ABC<
Main: str >ABC!!!<

ToUpper: source >def<
AppendString: source >DEF<
Main: str >DEF!!!<

ToUpper: source >ghi<
AppendString: source >GHI<
Main: str >GHI!!!<
```

En este ejemplo puede ver que cada método de extensión funciona de uno en uno para cada elemento de la recopilación de origen.

Lo que debe quedar claro de este ejemplo es que aunque hemos encadenado juntas consultas que producen recopilaciones, no se han materializado recopilaciones intermedias. En su lugar, cada elemento se pasa de un método diferido al siguiente. Esto tiene como resultado una superficie de memoria mucho menor que el enfoque que primero tomaría una matriz de cadenas y después crearía una segunda matriz de cadenas que se han convertido a mayúsculas y finalmente crearía una tercera matriz de cadenas en la que cada cadena tiene puntos de exclamación anexados.

El siguiente tema de este tutorial ilustra la materialización intermedia:

- Materialización intermedia (C#)

Vea también

- Tutorial: encadenar cadenas juntas (C#)

Materialización intermedia (C#)

18/03/2020 • 2 minutes to read • Edit Online

Si no tiene cuidado, en algunas situaciones puede alterar drásticamente el perfil de memoria y rendimiento de su aplicación causando una materialización prematura de las recopilaciones de sus consultas. Algunos operadores de consulta estándar provocan la materialización de su recopilación de origen antes de producir un elemento único. Por ejemplo, Enumerable.OrderBy primero recorre en iteración toda la recopilación de origen, después ordena todos los elementos y finalmente produce el primer elemento. Esto significa que resulta caro obtener el primer elemento de una recopilación ordenada; cada elemento posterior no es caro. Esto tiene sentido: sería imposible que el operador de consulta hiciera lo contrario.

Ejemplo

Este ejemplo altera el ejemplo anterior. El método `AppendString` llama a ToList antes de recorrer en iteración el origen. Esto provoca la materialización.

```
public static class LocalExtensions
{
    public static IEnumerable<string>
      ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
      AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        // the following statement materializes the source collection in a List<T>
        // before iterating through it
        foreach (string str in source.ToList())
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}
```

Este ejemplo produce el siguiente resultado:

```
ToUpper: source >abc<
ToUpper: source >def<
ToUpper: source >ghi<
AppendString: source >ABC<
Main: str >ABC!!!<

AppendString: source >DEF<
Main: str >DEF!!!<

AppendString: source >GHI<
Main: str >GHI!!!<
```

En este ejemplo puede ver que la llamada a ToList hace que `AppendString` enumere todo su origen antes de producir el primer elemento. Si el origen fuera una matriz grande, alteraría significativamente el perfil de memoria de la aplicación.

Los operadores de consulta estándar también se pueden encadenar juntos. Esto se ilustra en el tema final de este tutorial.

- Chaining Standard Query Operators Together (C#) (Encadenar juntos operadores de consulta estándar [C#])

Vea también

- Tutorial: encadenar cadenas juntas (C#)

Encadenar operadores de consulta estándar juntos (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este es el tema final del tutorial Tutorial: Chaining Queries Together (C#) (Tutorial: Encadenar consultas juntas [C#]).

Los operadores de consulta estándar también se pueden encadenar juntos. Por ejemplo, puede interponer el operador Enumerable.Where y también funciona de forma diferida. No materializa resultados intermedios.

Ejemplo

En este ejemplo, se llama al método Where antes de llamar a `ConvertCollectionToUpperCase`. El método Where funciona casi exactamente de la misma manera que los métodos diferidos utilizados en los ejemplos anteriores de este tutorial, `ConvertCollectionToUpperCase` y `AppendString`.

Una diferencia es que en este caso, el método Where recorre en iteración su recopilación de origen, determina que el primer elemento no pasa el predicado y después obtiene el siguiente elemento, que lo pasa. A continuación produce el segundo elemento.

No obstante, la idea básica es la misma: las recopilaciones intermedias no se materializan a menos que deban hacerlo.

Cuando se utilizan expresiones de consulta, se convierten en llamadas a operadores de consulta estándar y se aplican los mismos principios.

Todos los ejemplos de esta sección que consultan documentos XML abierto de Office utilizan el mismo principio. La ejecución aplazada y la evaluación diferida son algunos de los conceptos fundamentales que debe comprender para utilizar LINQ (y LINQ to XML) de forma efectiva.

```
public static class LocalExtensions
{
    public static IEnumerable<string>
      ConvertCollectionToUpperCase(this IEnumerable<string> source)
    {
        foreach (string str in source)
        {
            Console.WriteLine("ToUpper: source >{0}<", str);
            yield return str.ToUpper();
        }
    }

    public static IEnumerable<string>
      AppendString(this IEnumerable<string> source, string stringToAppend)
    {
        foreach (string str in source)
        {
            Console.WriteLine("AppendString: source >{0}<", str);
            yield return str + stringToAppend;
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        string[] stringArray = { "abc", "def", "ghi" };

        IEnumerable<string> q1 =
            from s in stringArray.ConvertCollectionToUpperCase()
            where s.CompareTo("D") >= 0
            select s;

        IEnumerable<string> q2 =
            from s in q1.AppendString("!!!")
            select s;

        foreach (string str in q2)
        {
            Console.WriteLine("Main: str >{0}<", str);
            Console.WriteLine();
        }
    }
}
```

Este ejemplo produce el siguiente resultado:

```
ToUpper: source >abc<
ToUpper: source >def<
AppendString: source >DEF<
Main: str >DEF!!!<

ToUpper: source >ghi<
AppendString: source >GHI<
Main: str >GHI!!!<
```

Forma de documentos WordprocessingML (C#)

18/03/2020 • 3 minutes to read • Edit Online

En este tema se presenta la forma XML de un documento WordprocessingML.

Formatos de Microsoft Office

El formato de archivo nativo para el sistema Microsoft Office 2007 es XML abierto de Office (comúnmente denominado XML abierto). XML abierto es un formato basado en XML que sigue la norma Ecma y que actualmente está sometido al proceso de estándares ISO-IEC. El lenguaje de marcado para los archivos de procesamiento de texto en XML abierto se denomina WordprocessingML. Este tutorial usa los archivos de origen WordprocessingML como datos para los ejemplos.

Si está utilizando Microsoft Office 2003, puede guardar documentos en el formato XML abierto de Office si ha instalado el Paquete de compatibilidad de Microsoft Office para los formatos de archivo Word, Excel y PowerPoint 2007.

Forma de los documentos WordprocessingML

Lo primero que hay que comprender es la forma de los documentos WordprocessingML. Un documento WordprocessingML contiene un elemento de Body (con el nombre `w:body`) que contiene los párrafos del documento. Cada párrafo contiene una o más ejecuciones de texto (denominadas `w:r`). Cada ejecución de texto contiene uno o más fragmentos de texto (con el nombre `w:t`).

A continuación se muestra un documento WordprocessingML muy sencillo:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<w:document
xmlns:ve="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:o="urn:schemas-microsoft-com:office:office"
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math"
xmlns:v="urn:schemas-microsoft-com:vml"
xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing"
xmlns:w10="urn:schemas-microsoft-com:office:word"
xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main"
xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml">
  <w:body>
    <w:p w:rsidR="00E22EB6"
         w:rsidRDefault="00E22EB6">
      <w:r>
        <w:t>This is a paragraph.</w:t>
      </w:r>
    </w:p>
    <w:p w:rsidR="00E22EB6"
         w:rsidRDefault="00E22EB6">
      <w:r>
        <w:t>This is another paragraph.</w:t>
      </w:r>
    </w:p>
  </w:body>
</w:document>
```

Este documento contiene dos párrafos. Contienen una ejecución de texto única y cada ejecución de texto contiene un solo fragmento de texto.

La forma más sencilla de ver el contenido de un documento WordprocessingML en forma XML es crear un documento mediante Microsoft Word, guardarlo y después ejecutar el siguiente programa que muestra XML en la consola.

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
const string documentRelationshipType =
  "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string wordmlNamespace =
  "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

using (Package wdPackage = Package.Open("SampleDoc.docx", FileMode.Open, FileAccess.Read))
{
    PackageRelationship relationship =
        wdPackage
        .GetRelationshipsByType(documentRelationshipType)
        .FirstOrDefault();
    if (relationship != null)
    {
        Uri documentUri =
            PackUriHelper.ResolvePartUri(
                new Uri("/", UriKind.Relative),
                relationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        //  Get the officeDocument part from the package.
        //  Load the XML in the part into an XDocument instance.
        XDocument xdoc =
            XDocument.Load(XmlReader.Create(documentPart.GetStream()));
        Console.WriteLine(xdoc.Root);
    }
}
```

Recursos externos

- Introducción a los formatos de archivo Office (2007) Open XML
- Información general de WordprocessingML
- Anatomía de un archivo WordProcessingML
- Introducción a WordprocessingML

Vea también

- Tutorial: Manipular contenido en un documento de WordprocessingML (C#)

Crear el documento de origen de Office Open XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

En este tema se muestra cómo crear el documento WordprocessingML XML abierto de Office que usan los otros ejemplos de este tutorial. Si sigue estas instrucciones, su resultado coincidirá con el resultado proporcionado en cada ejemplo.

No obstante, los ejemplos de este tutorial funcionarán con cualquier documento WordprocessingML válido.

Para crear el documento que se usa en este tutorial, debe tener Microsoft Office 2007 o posterior instalado o bien tener Microsoft Office 2003 con el paquete de compatibilidad de Microsoft Office para formatos de archivo de Word, Excel y PowerPoint 2007.

Crear el documento WordprocessingML

Para crear el documento WordprocessingML

1. Cree un nuevo documento de Microsoft Word.
2. Pegue el siguiente texto en el nuevo documento:

```
Parsing WordprocessingML with LINQ to XML

The following example prints to the console.

using System;

class Program {
    public static void Main(string[] args) {
        Console.WriteLine("Hello World");
    }
}

This example produces the following output:

Hello World
```

3. Dé formato a la primera línea con el estilo "Título 1".
4. Seleccione las líneas que contienen el código C#. La primera línea empieza con la palabra clave `using`. La última línea es la última llave de cierre. Dé formato a las líneas con la fuente Courier. Déles formato con un nuevo estilo y llame a ese nuevo estilo "Code".
5. Finalmente, seleccione toda la línea que contiene el resultado y déle formato con el estilo `Code`.
6. Guarde el documento y déle el nombre SampleDoc.docx.

> **NOTE**
>
> Si está usando Microsoft Word 2003, seleccione **Documento de Word 2007** en la lista desplegable **Guardar como tipo**.

Buscar el estilo de párrafo predeterminado (C#)

18/03/2020 • 2 minutes to read • Edit Online

La primera tarea del tutorial Manipular información en un documento WordprocessingML consiste en buscar el estilo predeterminado de los párrafos del documento.

Ejemplo

Description

El siguiente ejemplo abre un documento XML WordprocessingML abierto de Office, busca las partes del documento y del estilo del paquete y ejecuta una consulta que busca el nombre de estilo predeterminado. Para obtener información sobre los paquetes Office Open XML y las partes de las que constan, vea Información de los documentos de WordprocessingML de Office Open XML (C#).

La consulta busca un nodo con el nombre `w:style` que tiene un atributo con el nombre `w:type` con un valor de "paragraph" y también un atributo con el nombre `w:default` con un valor de "1". Puesto que sólo habrá un nodo XML con estos atributos, la consulta utiliza el operador Enumerable.First para convertir una recopilación en un singleton. A continuación obtiene el valor de un atributo con el nombre `w:styleId`.

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

Código

```
const string fileName = "SampleDoc.docx";

const string documentRelationshipType =
  "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string stylesRelationshipType =
  "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
const string wordmlNamespace =
  "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

XDocument xDoc = null;
XDocument styleDoc = null;

using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
{
    PackageRelationship docPackageRelationship =
      wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
    if (docPackageRelationship != null)
    {
        Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
          docPackageRelationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        //  Load the document XML in the part into an XDocument instance.
        xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

        //  Find the styles part. There will only be one.
        PackageRelationship styleRelation =
          documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
        if (styleRelation != null)
        {
            Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
            PackagePart stylePart = wdPackage.GetPart(styleUri);

            //  Load the style XML in the part into an XDocument instance.
            styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
        }
    }
}

// The following query finds all the paragraphs that have the default style.
string defaultStyle =
    (string)(
        from style in styleDoc.Root.Elements(w + "style")
        where (string)style.Attribute(w + "type") == "paragraph"&&
              (string)style.Attribute(w + "default") == "1"
              select style
    ).First().Attribute(w + "styleId");

Console.WriteLine("The default style is: {0}", defaultStyle);
```

Comentarios

Este ejemplo produce el siguiente resultado:

```
The default style is: Normal
```

Pasos siguientes

En el siguiente ejemplo, creará una consulta similar que busca todos los párrafos de un documento y sus estilos:

- Recuperar los párrafos y sus estilos (C#)

Recuperar los párrafos y sus estilos (C#)

18/03/2020 • 6 minutes to read • Edit Online

En este ejemplo, se escribe una consulta que recupera los nodos de párrafo de un documento WordprocessingML. También identifica el estilo de cada uno de los párrafos.

Esta consulta se basa en la consulta del ejemplo anterior, Buscar el estilo de párrafo predeterminado (C#), que recupera el estilo predeterminado de la lista de estilos. Esta información es necesaria para que la consulta pueda identificar el estilo de los párrafos que no tienen un estilo establecido explícitamente. Los estilos de párrafo se establecen mediante el elemento `w:pPr`; si un párrafo no contiene este elemento, se formatea con el estilo predeterminado.

Este tema explica la importancia de algunas partes de la consulta y luego muestra dicha consulta como parte de un ejemplo completo y funcional.

Ejemplo

El origen de la consulta para recuperar todos los párrafos de un documento y sus estilos es de la siguiente manera:

```
xDoc.Root.Element(w + "body").Descendants(w + "p")
```

Esta expresión es similar al origen de la consulta del ejemplo anterior, Buscar el estilo de párrafo predeterminado (C#). La diferencia principal radica en que usa el eje Descendants en lugar del eje Elements. La consulta usa el eje Descendants porque en los documentos que tienen secciones, lo párrafos no serán los secundarios directos del elemento de cuerpo; en su lugar, los párrafos estarán dos niveles por debajo en la jerarquía. Mediante el eje Descendants, el código funcionará independientemente de que el documento use secciones o no.

Ejemplo

La consulta usa una cláusula `let` para determinar el elemento que contiene el nodo de estilo. Si no hay ningún elemento, `styleNode` se establece en `null`:

```
let styleNode = para.Elements(w + "pPr").Elements(w + "pStyle").FirstOrDefault()
```

La cláusula `let` usa primero el eje Elements para encontrar todos los elementos llamados `pPr`, luego usa el método de extensión Elements para encontrar todos los elementos secundarios llamados `pStyle` y por último usa el operador de consulta estándar FirstOrDefault para convertir la colección en un singleton. Si la colección está vacía, `styleNode` se establece en `null`. Se trata de un método útil para buscar el nodo descendiente `pStyle`. Tenga en cuenta que si el nodo secundario `pPr` no existe, el código no produce errores generando una excepción; en su lugar, `styleNode` se establece en `null`, que es el comportamiento deseado de esta cláusula `let`.

La consulta proyecta una colección de un tipo anónimo con dos miembros, `StyleName` y `ParagraphNode`.

Ejemplo

Este ejemplo procesa un documento WordprocessingML, recuperando los nodos de párrafo a partir de dicho documento. También identifica el estilo de cada uno de los párrafos. Este ejemplo se basa en los ejemplos anteriormente vistos en este tutorial. En los comentarios del siguiente código se menciona dónde se encuentra la nueva consulta.

Encontrará instrucciones sobre cómo crear el documento de origen para este ejemplo en Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
const string fileName = "SampleDoc.docx";

const string documentRelationshipType =
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string stylesRelationshipType =
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
const string wordmlNamespace = "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

XDocument xDoc = null;
XDocument styleDoc = null;

using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
{
    PackageRelationship docPackageRelationship =
wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
    if (docPackageRelationship != null)
    {
        Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
docPackageRelationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        //  Load the document XML in the part into an XDocument instance.
        xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

        //  Find the styles part. There will only be one.
        PackageRelationship styleRelation =
documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
        if (styleRelation != null)
        {
            Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
            PackagePart stylePart = wdPackage.GetPart(styleUri);

            //  Load the style XML in the part into an XDocument instance.
            styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
        }
    }
}

string defaultStyle =
    (string)(
        from style in styleDoc.Root.Elements(w + "style")
        where (string)style.Attribute(w + "type") == "paragraph"&&
              (string)style.Attribute(w + "default") == "1"
              select style
    ).First().Attribute(w + "styleId");

// Following is the new query that finds all paragraphs in the
// document and their styles.
var paragraphs =
    from para in xDoc
                 .Root
                 .Element(w + "body")
                 .Descendants(w + "p")
    let styleNode = para
                    .Elements(w + "pPr")
                    .Elements(w + "pStyle")
                    .FirstOrDefault()
    select new
    {
        ParagraphNode = para,
```

```
        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    };

foreach (var p in paragraphs)
    Console.WriteLine("StyleName:{0}", p.StyleName);
```

Este ejemplo genera el siguiente resultado cuando se aplica al documento descrito en Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

```
StyleName:Heading1
StyleName:Normal
StyleName:Normal
StyleName:Normal
StyleName:Code
StyleName:Code
StyleName:Code
StyleName:Code
StyleName:Code
StyleName:Code
StyleName:Code
StyleName:Normal
StyleName:Normal
StyleName:Normal
StyleName:Code
```

Pasos siguientes

En el tema siguiente, Recuperar el texto de los párrafos (C#), podrá crear una consulta para recuperar el texto de los párrafos.

Vea también

- Tutorial: Manipular contenido en un documento de WordprocessingML (C#)

Recuperar el texto de los párrafos (C#)

18/03/2020 • 5 minutes to read • Edit Online

Este ejemplo se basa en el ejemplo anterior, Recuperar los párrafos y sus estilos (C#). El nuevo ejemplo recupera el texto de cada párrafo como una cadena.

Para ello, incorpora una consulta nueva que lleva a cabo una iteración a lo largo de la colección de tipos anónimos y proyecta una nueva colección de un tipo anónimo incluyendo a un nuevo miembro, `Text`. Utiliza el operador de consultas estándar Aggregate para concatenar varias cadenas en una sola cadena.

Esta técnica (es decir, primeramente proyectar una colección de un tipo anónimo para después utilizarla para proyectarla en una colección nueva de un tipo anónimo) es un común y resulta muy útil. Se podría haber escrito esta consulta sin que se realice una proyección en el primer tipo anónimo. Sin embargo, debido a la evaluación aplazada, no se utilizaría mucha potencia de procesamiento adicional en caso de utilizarse. Esta técnica crea objetos de menor duración en la pila de memoria, pero esto no reduce de forma significativa el rendimiento.

Por supuesto, sería posible escribir una única consulta que contuviese la funcionalidad necesaria para recuperar los párrafos, el estilo de cada uno de ellos y su texto. Sin embargo, a menudo resulta útil dividir una consulta más compleja en múltiples consultas, ya que el código resultante será más modular y fácil de mantener. Y lo que es más, si necesitara reutilizar una parte de esa consulta, resultaría más sencillo refactorizarla si las consultas están escritas de esta forma.

Estas consultas, que están encadenadas unas a otras, usan el modelo de procesamiento que se describe detalladamente en el tema Tutorial: Chaining Queries Together (C#) (Tutorial: Encadenar consultas juntas (C#)).

Ejemplo

Este ejemplo procesa un documento WordprocessingML, determinando el nodo elemento, el nombre del estilo y el texto de cada párrafo. Este ejemplo se basa en los ejemplos anteriormente vistos en este tutorial. En los comentarios del siguiente código se menciona dónde se encuentra la nueva consulta.

Para obtener instrucciones sobre cómo crear el documento de origen de este ejemplo, vea Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
const string fileName = "SampleDoc.docx";

const string documentRelationshipType =
  "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string stylesRelationshipType =
  "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
const string wordmlNamespace =
  "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

XDocument xDoc = null;
XDocument styleDoc = null;

using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
{
    PackageRelationship docPackageRelationship =
      wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
    if (docPackageRelationship != null)
    {
```

```
        Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
          docPackageRelationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        //  Load the document XML in the part into an XDocument instance.
        xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

        //  Find the styles part. There will only be one.
        PackageRelationship styleRelation =
          documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
        if (styleRelation != null)
        {
            Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
            PackagePart stylePart = wdPackage.GetPart(styleUri);

            //  Load the style XML in the part into an XDocument instance.
            styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
        }
    }
}

string defaultStyle =
    (string)(
        from style in styleDoc.Root.Elements(w + "style")
        where (string)style.Attribute(w + "type") == "paragraph"&&
              (string)style.Attribute(w + "default") == "1"
              select style
    ).First().Attribute(w + "styleId");

// Find all paragraphs in the document.
var paragraphs =
    from para in xDoc
                 .Root
                 .Element(w + "body")
                 .Descendants(w + "p")
    let styleNode = para
                    .Elements(w + "pPr")
                    .Elements(w + "pStyle")
                    .FirstOrDefault()
    select new
    {
        ParagraphNode = para,
        StyleName = styleNode != null ?
            (string)styleNode.Attribute(w + "val") :
            defaultStyle
    };

// Following is the new query that retrieves the text of
// each paragraph.
var paraWithText =
    from para in paragraphs
    select new
    {
        ParagraphNode = para.ParagraphNode,
        StyleName = para.StyleName,
        Text = para
               .ParagraphNode
               .Elements(w + "r")
               .Elements(w + "t")
               .Aggregate(
                   new StringBuilder(),
                   (s, i) => s.Append((string)i),
                   s => s.ToString()
               )
    };

foreach (var p in paraWithText)
    Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
```

Este ejemplo genera el siguiente resultado cuando se aplica al documento descrito en Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

```
StyleName:Heading1 >Parsing WordprocessingML with LINQ to XML<
StyleName:Normal ><
StyleName:Normal >The following example prints to the console.<
StyleName:Normal ><
StyleName:Code >using System;<
StyleName:Code ><
StyleName:Code >class Program {<
StyleName:Code >    public static void (string[] args) {<
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >    }<
StyleName:Code >}<
StyleName:Normal ><
StyleName:Normal >This example produces the following output:<
StyleName:Normal ><
StyleName:Code >Hello World<
```

Pasos siguientes

El siguiente ejemplo muestra cómo utilizar un método de extensión, en vez de Aggregate, para concatenar múltiples cadenas en una única cadena.

- Refactorizar mediante un método de extensión (C#)

Vea también

- Tutorial: Manipular contenido en un documento de WordprocessingML (C#)
- Ejecución aplazada y evaluación diferida en LINQ to XML (C#)

Refactorizar mediante un método de extensión (C#)

18/03/2020 • 6 minutes to read • Edit Online

Este ejemplo se basa en el ejemplo anterior, Recuperar el texto de los párrafos (C#), mediante la refactorización de la concatenación de cadenas con una función pura que se implementa como método de extensión.

El ejemplo anterior usaba el operador de consultas estándar Aggregate para concatenar varias cadenas en una sola. Sin embargo, resulta más eficaz escribir un método de extensión a tal efecto, ya que la consulta resultante es más pequeña y más simple.

Ejemplo

Este ejemplo procesa un documento WordprocessingML, recuperando los párrafos, así como el estilo y el texto de cada párrafo. Este ejemplo se basa en los ejemplos anteriormente vistos en este tutorial.

El ejemplo contiene varias sobrecargas del método `StringConcatenate`.

Encontrará instrucciones sobre cómo crear el documento de origen para este ejemplo en Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}
```

Ejemplo

Hay cuatro sobrecargas del método `StringConcatenate`. Una sobrecarga simplemente toma una colección de cadenas y devuelve una sola cadena. Otra sobrecarga puede tomar una colección de cualquier tipo y un delegado que se proyecta de un singleton de la colección a una cadena. Hay dos sobrecargas más que permiten especificar una cadena de separación.

El código siguiente usa las cuatro sobrecargas.

```
string[] numbers = { "one", "two", "three" };

Console.WriteLine("{0}", numbers.StringConcatenate());
Console.WriteLine("{0}", numbers.StringConcatenate(":"));

int[] intNumbers = { 1, 2, 3 };
Console.WriteLine("{0}", intNumbers.StringConcatenate(i => i.ToString()));
Console.WriteLine("{0}", intNumbers.StringConcatenate(i => i.ToString(), ":"));
```

Este ejemplo produce el siguiente resultado:

```
onetwothree
one:two:three:
123
1:2:3:
```

Ejemplo

Ahora, el ejemplo puede modificarse para aprovechar el nuevo método de extensión:

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
          "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
              wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
                  docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);
```

```
                //  Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                //  Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                  documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri =
                      PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    //  Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                      (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");

        // Find all paragraphs in the document.
        var paragraphs =
            from para in xDoc
                         .Root
                         .Element(w + "body")
                         .Descendants(w + "p")
            let styleNode = para
                            .Elements(w + "pPr")
                            .Elements(w + "pStyle")
                            .FirstOrDefault()
            select new
            {
                ParagraphNode = para,
                StyleName = styleNode != null ?
                    (string)styleNode.Attribute(w + "val") :
                    defaultStyle
            };

        // Retrieve the text of each paragraph.
        var paraWithText =
            from para in paragraphs
            select new
            {
                ParagraphNode = para.ParagraphNode,
                StyleName = para.StyleName,
                Text = para
                       .ParagraphNode
                       .Elements(w + "r")
                       .Elements(w + "t")
                       .StringConcatenate(e => (string)e)
            };

        foreach (var p in paraWithText)
            Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
    }
}
```

Este ejemplo genera el siguiente resultado cuando se aplica al documento descrito en Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

```
StyleName:Heading1 >Parsing WordprocessingML with LINQ to XML<
StyleName:Normal ><
StyleName:Normal >The following example prints to the console.<
StyleName:Normal ><
StyleName:Code >using System;<
StyleName:Code ><
StyleName:Code >class Program {<
StyleName:Code >    public static void (string[] args) {<
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >    }<
StyleName:Code >}<
StyleName:Normal ><
StyleName:Normal >This example produces the following output:<
StyleName:Normal ><
StyleName:Code >Hello World<
```

Tenga en cuenta que esta refactorización es una variante de refactorizar en una función pura. El siguiente tema presentará la idea de factorizar en funciones puras con mayor detalle.

Pasos siguientes

El siguiente ejemplo muestra cómo refactorizar este código de otra forma, mediante las funciones puras:

- Refactorizar mediante una función pura (Visual Basic)

Vea también

- Tutorial: Manipular contenido en un documento de WordprocessingML (C#)
- Refactorizar en funciones puras (C#)

Refactorización mediante una función pura (C#)

18/03/2020 • 4 minutes to read • Edit Online

En el siguiente ejemplo se refactoriza el ejemplo anterior, Refactorizar mediante un método de extensión (C#), para usar una función pura. En este ejemplo, el código encargado de buscar el texto de un párrafo pasa a formar parte del método estático puro `ParagraphText`.

Ejemplo

Este ejemplo procesa un documento WordprocessingML, recuperando los nodos de párrafo a partir de dicho documento. También identifica el estilo de cada uno de los párrafos. Este ejemplo se basa en los ejemplos anteriormente vistos en este tutorial. En los comentarios del siguiente código se menciona dónde se encuentra el código refactorizado.

Para obtener instrucciones sobre cómo crear el documento de origen de este ejemplo, vea Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
```

```
    // This is a new method that assembles the paragraph text.
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
                .Elements(w + "r")
                .Elements(w + "t")
                .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
          "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
              wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
                  docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);

                //  Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                //  Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                  documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri = PackUriHelper.ResolvePartUri(documentUri,
                      styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    //  Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                      (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");

        // Find all paragraphs in the document.
        var paragraphs =
            from para in xDoc
                         .Root
                         .Element(w + "body")
                         .Descendants(w + "p")
            let styleNode = para
                            .Elements(w + "pPr")
```

```
                            .Elements(w + "pStyle")
                            .FirstOrDefault()
            select new
            {
                ParagraphNode = para,
                StyleName = styleNode != null ?
                    (string)styleNode.Attribute(w + "val") :
                    defaultStyle
            };

        // Retrieve the text of each paragraph.
        var paraWithText =
            from para in paragraphs
            select new
            {
                ParagraphNode = para.ParagraphNode,
                StyleName = para.StyleName,
                Text = ParagraphText(para.ParagraphNode)
            };

        foreach (var p in paraWithText)
            Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
    }
}
```

Este ejemplo genera la misma salida que antes de la refactorización:

```
StyleName:Heading1 >Parsing WordprocessingML with LINQ to XML<
StyleName:Normal ><
StyleName:Normal >The following example prints to the console.<
StyleName:Normal ><
StyleName:Code >using System;<
StyleName:Code ><
StyleName:Code >class Program {<
StyleName:Code >    public static void (string[] args) {<
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >    }<
StyleName:Code >}<
StyleName:Normal ><
StyleName:Normal >This example produces the following output:<
StyleName:Normal ><
StyleName:Code >Hello World<
```

Pasos siguientes

El siguiente ejemplo muestra cómo proyectar un XML con un aspecto diferente:

- Proyectar XML en una forma distinta (C#)

Vea también

- Tutorial: Manipular contenido en un documento de WordprocessingML (C#)
- Refactorizar mediante un método de extensión (C#)
- Refactorizar en funciones puras (C#)

Proyección de XML en una forma distinta (C#)

18/03/2020 • 4 minutes to read • Edit Online

Este tema muestra un ejemplo de cómo proyectar un XML cuya apariencia es diferente a la del XML de origen.

Muchas de las transformación típicas de un XML consisten en consultas encadenadas, como ocurre en este ejemplo. Es habitual comenzar con una cierta forma de XML, proyectar los resultados intermedios como colecciones de tipos anónimos o con nombre y, finalmente, proyectar de nuevo los resultados en XML, cuya forma es completamente diferente a la del XML de origen.

Ejemplo

Este ejemplo procesa un documento WordprocessingML, recuperando los nodos de párrafo a partir de dicho documento. También identifica el estilo y el texto de cada uno de los párrafos. Por último, el ejemplo proyecta el XML con una forma diferente. Este ejemplo se basa en los ejemplos anteriormente vistos en este tutorial. La nueva instrucción encargada de la proyección está señalada en los comentarios del código siguiente:

Para obtener instrucciones sobre cómo crear el documento de origen de este ejemplo, vea Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}
```

```
}

class Program
{
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
               .Elements(w + "r")
               .Elements(w + "t")
               .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
          "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
              wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
                  docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);

                //  Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                //  Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                  documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri =
                      PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    //  Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                      (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");

        // Find all paragraphs in the document.
        var paragraphs =
            from para in xDoc
                         .Root
                         .Element(w + "body")
```

```
                        .Descendants(w + "p")
            let styleNode = para
                            .Elements(w + "pPr")
                            .Elements(w + "pStyle")
                            .FirstOrDefault()
            select new
            {
                ParagraphNode = para,
                StyleName = styleNode != null ?
                    (string)styleNode.Attribute(w + "val") :
                    defaultStyle
            };

        // Retrieve the text of each paragraph.
        var paraWithText =
            from para in paragraphs
            select new
            {
                ParagraphNode = para.ParagraphNode,
                StyleName = para.StyleName,
                Text = ParagraphText(para.ParagraphNode)
            };

        // The following is the new code that projects XML in a new shape.
        XElement root = new XElement("Root",
            from p in paraWithText
            select new XElement("Paragraph",
                new XElement("StyleName", p.StyleName),
                new XElement("Text", p.Text)
            )
        );

        Console.WriteLine(root);
    }
}
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <Paragraph>
    <StyleName>Heading1</StyleName>
    <Text>Parsing WordprocessingML with LINQ to XML</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text>The following example prints to the console.</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>using System;</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>class Program {</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>    public static void (string[] args) {</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>        Console.WriteLine("Hello World");</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>    }</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>}</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text>This example produces the following output:</Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Normal</StyleName>
    <Text></Text>
  </Paragraph>
  <Paragraph>
    <StyleName>Code</StyleName>
    <Text>Hello World</Text>
  </Paragraph>
</Root>
```

Pasos siguientes

En el siguiente ejemplo, realizará una consulta para buscar todo el texto de un documento de Word:

- Buscar texto en documentos de Word (C#)

Buscar texto en documentos de Word (C#)

18/03/2020 • 7 minutes to read • Edit Online

En este tema se amplían las consultas vistas anteriormente para conseguir algo muy útil: encontrar todas las apariciones de una cierta cadena en el documento.

Ejemplo

Este ejemplo procesa un documento WordprocessingML, con el objetivo de encontrar todas las apariciones de una porción de texto en particular dentro del documento. Para ello, utilizaremos una consulta que encuentre la cadena "Hello". Este ejemplo se basa en los ejemplos anteriormente vistos en este tutorial. En los comentarios del siguiente código se menciona dónde se encuentra la nueva consulta.

Para obtener instrucciones sobre cómo crear el documento de origen de este ejemplo, vea Creating the Source Office Open XML Document (C#) (Creación del documento de origen de Office Open XML (C#)).

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
    public static string ParagraphText(XElement e)
```

```
    {
        XNamespace w = e.Name.Namespace;
        return e
               .Elements(w + "r")
               .Elements(w + "t")
               .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
          "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace =
          "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
              wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
                  docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);

                //  Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                //  Find the styles part. There will only be one.
                PackageRelationship styleRelation =
                  documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri =
                      PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    //  Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                      (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");

        // Find all paragraphs in the document.
        var paragraphs =
            from para in xDoc
                         .Root
                         .Element(w + "body")
                         .Descendants(w + "p")
            let styleNode = para
                            .Elements(w + "pPr")
                            .Elements(w + "pStyle")
                            .FirstOrDefault()
```

```
            select new
            {
                ParagraphNode = para,
                StyleName = styleNode != null ?
                    (string)styleNode.Attribute(w + "val") :
                    defaultStyle
            };

        // Retrieve the text of each paragraph.
        var paraWithText =
            from para in paragraphs
            select new
            {
                ParagraphNode = para.ParagraphNode,
                StyleName = para.StyleName,
                Text = ParagraphText(para.ParagraphNode)
            };

        // Following is the new query that retrieves all paragraphs
        // that have specific text in them.
        var helloParagraphs =
            from para in paraWithText
            where para.Text.Contains("Hello")
            select new
            {
                ParagraphNode = para.ParagraphNode,
                StyleName = para.StyleName,
                Text = para.Text
            };

        foreach (var p in helloParagraphs)
            Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
    }
}
```

Este ejemplo produce el siguiente resultado:

```
StyleName:Code >        Console.WriteLine("Hello World");<
StyleName:Code >Hello World<
```

Por supuesto, puede modificar la búsqueda de forma que encuentre aquellas líneas con un estilo en particular. La siguiente consulta encuentra todas las líneas en blanco que tienen el estilo Code:

```
public static class LocalExtensions
{
    public static string StringConcatenate(this IEnumerable<string> source)
    {
        StringBuilder sb = new StringBuilder();
        foreach (string s in source)
            sb.Append(s);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item));
        return sb.ToString();
    }

    public static string StringConcatenate(this IEnumerable<string> source, string separator)
    {
        StringBuilder sb = new StringBuilder();
```

```
        foreach (string s in source)
            sb.Append(s).Append(separator);
        return sb.ToString();
    }

    public static string StringConcatenate<T>(this IEnumerable<T> source,
        Func<T, string> func, string separator)
    {
        StringBuilder sb = new StringBuilder();
        foreach (T item in source)
            sb.Append(func(item)).Append(separator);
        return sb.ToString();
    }
}

class Program
{
    public static string ParagraphText(XElement e)
    {
        XNamespace w = e.Name.Namespace;
        return e
               .Elements(w + "r")
               .Elements(w + "t")
               .StringConcatenate(element => (string)element);
    }

    static void Main(string[] args)
    {
        const string fileName = "SampleDoc.docx";

        const string documentRelationshipType =
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
        const string stylesRelationshipType =
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
        const string wordmlNamespace = "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
        XNamespace w = wordmlNamespace;

        XDocument xDoc = null;
        XDocument styleDoc = null;

        using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
        {
            PackageRelationship docPackageRelationship =
wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
            if (docPackageRelationship != null)
            {
                Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
docPackageRelationship.TargetUri);
                PackagePart documentPart = wdPackage.GetPart(documentUri);

                //  Load the document XML in the part into an XDocument instance.
                xDoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

                //  Find the styles part. There will only be one.
                PackageRelationship styleRelation =
documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
                if (styleRelation != null)
                {
                    Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
                    PackagePart stylePart = wdPackage.GetPart(styleUri);

                    //  Load the style XML in the part into an XDocument instance.
                    styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));
                }
            }
        }

        string defaultStyle =
            (string)(
```

```
                from style in styleDoc.Root.Elements(w + "style")
                where (string)style.Attribute(w + "type") == "paragraph" &&
                      (string)style.Attribute(w + "default") == "1"
                select style
            ).First().Attribute(w + "styleId");

        // Find all paragraphs in the document.
        var paragraphs =
            from para in xDoc
                         .Root
                         .Element(w + "body")
                         .Descendants(w + "p")
            let styleNode = para
                            .Elements(w + "pPr")
                            .Elements(w + "pStyle")
                            .FirstOrDefault()
            select new
            {
                ParagraphNode = para,
                StyleName = styleNode != null ?
                    (string)styleNode.Attribute(w + "val") :
                    defaultStyle
            };

        // Retrieve the text of each paragraph.
        var paraWithText =
            from para in paragraphs
            select new
            {
                ParagraphNode = para.ParagraphNode,
                StyleName = para.StyleName,
                Text = ParagraphText(para.ParagraphNode)
            };

        // Retrieve all paragraphs that have no text and are styled Code.
        var blankCodeParagraphs =
            from para in paraWithText
            where para.Text == "" && para.StyleName == "Code"
            select new
            {
                ParagraphNode = para.ParagraphNode,
                StyleName = para.StyleName,
                Text = para.Text
            };

        foreach (var p in blankCodeParagraphs)
            Console.WriteLine("StyleName:{0} >{1}<", p.StyleName, p.Text);
    }
}
```

Este ejemplo produce el siguiente resultado:

```
StyleName:Code ><
```

Por supuesto, se podría mejorar este ejemplo de diversas maneras. Por ejemplo, podríamos utilizar expresiones regulares para buscar texto, podríamos recorrer todos los archivos de Word de un directorio en particular, etc.

Observe que el rendimiento de este ejemplo es prácticamente igual que si se hubiese escrito en una única consulta. Dado que cada consulta se ha implementado utilizando el modelo diferido, éstas no proporcionarán sus resultados hasta que se itere la consulta. Para más información sobre la ejecución y evaluación diferidas, vea Ejecución aplazada y evaluación diferida en LINQ to XML (C#).

Pasos siguientes

En la sección siguiente se proporciona más información acerca de los documentos WordprocessingML:

- Detalles de los documentos de WordprocessingML de Office Open XML (C#)

Vea también

- Tutorial: Manipular contenido en un documento de WordprocessingML (C#)
- Refactorizar mediante una función pura (C#)
- Ejecución aplazada y evaluación diferida en LINQ to XML (C#)

Documento de WordprocessingML con estilos

18/03/2020 • 4 minutes to read • Edit Online

Los documentos WordprocessingML más complicados tienen párrafos formateados con estilos.

Resultan útiles algunas indicaciones acerca de la creación de documentos WordprocessingML. Los documentos WordprocessingML se almacenan en paquetes. Los paquetes tienen varias partes. Éstas tienen un significado explícito si se usan el contexto de los paquetes; básicamente, las partes son archivos que se comprimen para formar un paquete. Si un documento contiene párrafos que se formatean con estilos, habrá una parte de documento que contendrá párrafos con estilos aplicados a éstos. También habrá una parte de estilo que contendrá los estilos a los que hace referencia el documento.

Cuando se obtiene acceso a los paquetes, es importante hacerlo mediante la relación entre las partes, en lugar de usar una ruta de acceso arbitraria. Este problema se encuentra fuera del ámbito del tutorial Manipular contenido en un documento de WordprocessingML, aunque los programas de ejemplo incluidos en este tutorial demuestran el método correcto.

Documento que usa estilos

El ejemplo de WordML presentado en el tema Forma de documentos de WordprocessingML es muy simple. El documento siguiente es más complicado: tiene párrafos que están formateados con estilos. La forma más fácil de ver el XML que forma un documento de Office Open XML consiste en ejecutar el Example that Outputs Office Open XML Document Parts (C#) (Ejemplo en el que se generan partes de un documento de Office Open XML (C#)).

En el documento siguiente, el primer párrafo tiene el estilo `Heading1`. Hay varios párrafos que tienen el estilo predeterminado. Asimismo, existen diversos párrafos con el estilo `Code`. Debido a esta complejidad relativa, resulta más interesante analizar este documento con LINQ to XML.

En los párrafos con estilos no predeterminados, los elementos de párrafo tienen un elemento secundario llamado `w:pPr`, que a su vez tiene un elemento secundario `w:pStyle`. El elemento tiene un atributo, `w:val`, que contiene el nombre de estilo. Si el párrafo tiene el estilo predeterminado, significa que el elemento de párrafo no tiene un elemento secundario `w:p.Pr`.

```
<?xml version="1.0" encoding="utf-8"?>
<w:document
    xmlns:ve="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:o="urn:schemas-microsoft-com:office:office"
    xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
    xmlns:m="http://schemas.openxmlformats.org/officeDocument/2006/math"
    xmlns:v="urn:schemas-microsoft-com:vml"
    xmlns:wp="http://schemas.openxmlformats.org/drawingml/2006/wordprocessingDrawing"
    xmlns:w10="urn:schemas-microsoft-com:office:word"
    xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main"
    xmlns:wne="http://schemas.microsoft.com/office/word/2006/wordml">
  <w:body>
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
      <w:pPr>
        <w:pStyle w:val="Heading1" />
      </w:pPr>
      <w:r>
        <w:t>Parsing WordprocessingML with LINQ to XML</w:t>
      </w:r>
    </w:p>
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0">
      <w:r>
```

```
    <w:t>The following example prints to the console.</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r>
    <w:t>using System;</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t>class Program {</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t xml:space="preserve">    public static void </w:t>
  </w:r>
  <w:smartTag w:uri="urn:schemas-microsoft-com:office:smarttags" w:element="place">
    <w:r w:rsidRPr="00876F34">
      <w:t>Main</w:t>
    </w:r>
  </w:smartTag>
  <w:r w:rsidRPr="00876F34">
    <w:t>(string[] args) {</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t xml:space="preserve"        Console.WriteLine("Hello World");</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t xml:space="preserve">    }</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRPr="00876F34" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
  <w:pPr>
    <w:pStyle w:val="Code" />
  </w:pPr>
  <w:r w:rsidRPr="00876F34">
    <w:t>}</w:t>
  </w:r>
</w:p>
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />
<w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0">
  <w:r>
    <w:t>This example produces the following output:</w:t>
```

```
        <w:t>This example produces the following output.</w:t>
      </w:r>
    </w:p>
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" />
    <w:p w:rsidR="00A75AE0" w:rsidRDefault="00A75AE0" w:rsidP="006027C7">
      <w:pPr>
        <w:pStyle w:val="Code" />
      </w:pPr>
      <w:r>
        <w:t>Hello World</w:t>
      </w:r>
    </w:p>
    <w:sectPr w:rsidR="00A75AE0" w:rsidRDSect="00A75AE0">
      <w:pgSz w:w="12240" w:h="15840" />
      <w:pgMar w:top="1440" w:right="1800" w:bottom="1440" w:left="1800" w:header="720" w:footer="720"
w:gutter="0" />
      <w:cols w:space="720" />
      <w:docGrid w:linePitch="360" />
    </w:sectPr>
  </w:body>
</w:document>
```

Componente de estilo de un documento de WordprocessingML

18/03/2020 • 3 minutes to read • Edit Online

En este tema se muestra un ejemplo de la parte de estilo del documento WordprocessingML XML abierto de Office.

Ejemplo

El siguiente ejemplo es el XML que constituye la parte de estilo de un documento WordprocessingML XML abierto de Office.

El estilo de párrafo predeterminado tiene un elemento con la siguiente etiqueta de apertura:

```
<w:style w:type="paragraph" w:default="1" w:styleId="Normal">
```

Debe conocer esta información cuando escriba la consulta para buscar el identificador de estilo predeterminado, de forma que la consulta pueda identificar el estilo de los párrafos que tienen el estilo predeterminado.

Tenga en cuenta que esos documentos son muy simples en comparación con los documentos típicos que Microsoft Word genera. En muchos casos, Word guarda gran cantidad de información adicional, formato adicional y metadatos. Asimismo, Word no da formato a las líneas para que sean fácilmente legibles, como en este ejemplo; en su lugar, el XML se guarda sin sangría. No obstante, todos los documentos WordprocessingML comparten la misma forma XML básica. Por eso, las consultas que se presentan en este tutorial funcionarán con documentos más complicados.

```
<?xml version="1.0" encoding="utf-8"?>
<w:styles
    xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships"
    xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main">
  <w:docDefaults>
    <w:rPrDefault>
      <w:rPr>
        <w:rFonts
            w:ascii="Times New Roman"
            w:eastAsia="Times New Roman"
            w:hAnsi="Times New Roman"
            w:cs="Times New Roman" />
        <w:sz w:val="22" />
        <w:szCs w:val="22" />
        <w:lang w:val="en-US" w:eastAsia="en-US" w:bidi="ar-SA" />
      </w:rPr>
    </w:rPrDefault>
    <w:pPrDefault />
  </w:docDefaults>
  <w:style w:type="paragraph" w:default="1" w:styleId="Normal">
    <w:name w:val="Normal" />
    <w:qFormat />
    <w:rPr>
      <w:sz w:val="24" />
      <w:szCs w:val="24" />
    </w:rPr>
  </w:style>
  <w:style w:type="paragraph" w:styleId="Heading1">
    <w:name w:val="heading 1" />
    <w:basedOn w:val="Normal" />
    <w:next w:val="Normal" />
```

```
    <w:next w:val="Normal" />
    <w:link w:val="Heading1Char" />
    <w:uiPriority w:val="99" />
    <w:qFormat />
    <w:rsid w:val="006027C7" />
    <w:pPr>
      <w:keepNext />
      <w:spacing w:before="240" w:after="60" />
      <w:outlineLvl w:val="0" />
    </w:pPr>
    <w:rPr>
      <w:rFonts w:ascii="Arial" w:hAnsi="Arial" w:cs="Arial" />
      <w:b />
      <w:bCs />
      <w:kern w:val="32" />
      <w:sz w:val="32" />
      <w:szCs w:val="32" />
    </w:rPr>
  </w:style>
  <w:style w:type="character" w:default="1" w:styleId="DefaultParagraphFont">
    <w:name w:val="Default Paragraph Font" />
    <w:uiPriority w:val="99" />
    <w:semiHidden />
  </w:style>
  <w:style w:type="table" w:default="1" w:styleId="TableNormal">
    <w:name w:val="Normal Table" />
    <w:uiPriority w:val="99" />
    <w:semiHidden />
    <w:unhideWhenUsed />
    <w:qFormat />
    <w:tblPr>
      <w:tblInd w:w="0" w:type="dxa" />
      <w:tblCellMar>
        <w:top w:w="0" w:type="dxa" />
        <w:left w:w="108" w:type="dxa" />
        <w:bottom w:w="0" w:type="dxa" />
        <w:right w:w="108" w:type="dxa" />
      </w:tblCellMar>
    </w:tblPr>
  </w:style>
  <w:style w:type="numbering" w:default="1" w:styleId="NoList">
    <w:name w:val="No List" />
    <w:uiPriority w:val="99" />
    <w:semiHidden />
    <w:unhideWhenUsed />
  </w:style>
  <w:style w:type="character" w:customStyle="1" w:styleId="Heading1Char">
    <w:name w:val="Heading 1 Char" />
    <w:basedOn w:val="DefaultParagraphFont" />
    <w:link w:val="Heading1" />
    <w:uiPriority w:val="9" />
    <w:rsid w:val="009765E3" />
    <w:rPr>
      <w:rFonts
          w:asciiTheme="majorHAnsi"
          w:eastAsiaTheme="majorEastAsia"
          w:hAnsiTheme="majorHAnsi"
          w:cstheme="majorBidi" />
      <w:b />
      <w:bCs />
      <w:kern w:val="32" />
      <w:sz w:val="32" />
      <w:szCs w:val="32" />
    </w:rPr>
  </w:style>
  <w:style w:type="paragraph" w:customStyle="1" w:styleId="Code">
    <w:name w:val="Code" />
    <w:aliases w:val="c" />
    <w:uiPriority w:val="99" />
    <w:rsid w:val="006027C7" />
```

```
    <w:rsid w:val="00002FC7" />
    <w:pPr>
      <w:spacing w:after="60" w:line="300" w:lineRule="exact" />
    </w:pPr>
    <w:rPr>
      <w:rFonts w:ascii="Courier New" w:hAnsi="Courier New" />
      <w:noProof />
      <w:color w:val="000080" />
      <w:sz w:val="20" />
      <w:szCs w:val="20" />
    </w:rPr>
  </w:style>
</w:styles>
```

Ejemplo que da como resultado partes de un documento de Office Open XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema muestra cómo abrir un documento XML abierto de Office y obtener acceso a sus partes.

Ejemplo

El siguiente ejemplo abre un documento XML abierto de Office e imprime las partes de documento y estilo de la consola.

Este ejemplo utiliza las clases que se encuentran en el ensamblado WindowsBase. Utiliza los tipos del espacio de nombres System.IO.Packaging.

```
const string fileName = "SampleDoc.docx";

const string documentRelationshipType =
  "http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocument";
const string stylesRelationshipType =
  "http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles";
const string wordmlNamespace =
  "http://schemas.openxmlformats.org/wordprocessingml/2006/main";
XNamespace w = wordmlNamespace;

using (Package wdPackage = Package.Open(fileName, FileMode.Open, FileAccess.Read))
{
    PackageRelationship docPackageRelationship =
      wdPackage.GetRelationshipsByType(documentRelationshipType).FirstOrDefault();
    if (docPackageRelationship != null)
    {
        Uri documentUri = PackUriHelper.ResolvePartUri(new Uri("/", UriKind.Relative),
          docPackageRelationship.TargetUri);
        PackagePart documentPart = wdPackage.GetPart(documentUri);

        //  Load the document XML in the part into an XDocument instance.
        XDocument xdoc = XDocument.Load(XmlReader.Create(documentPart.GetStream()));

        Console.WriteLine("TargetUri:{0}", docPackageRelationship.TargetUri);
        Console.WriteLine("================================================================");
        Console.WriteLine(xdoc.Root);
        Console.WriteLine();

        //  Find the styles part. There will only be one.
        PackageRelationship styleRelation =
          documentPart.GetRelationshipsByType(stylesRelationshipType).FirstOrDefault();
        if (styleRelation != null)
        {
            Uri styleUri = PackUriHelper.ResolvePartUri(documentUri, styleRelation.TargetUri);
            PackagePart stylePart = wdPackage.GetPart(styleUri);

            //  Load the style XML in the part into an XDocument instance.
            XDocument styleDoc = XDocument.Load(XmlReader.Create(stylePart.GetStream()));

            Console.WriteLine("TargetUri:{0}", styleRelation.TargetUri);
            Console.WriteLine("================================================================");
            Console.WriteLine(styleDoc.Root);
            Console.WriteLine();
        }
    }
}
```

Diferencias entre la modificación del árbol XML en memoria y Construcción funcional (LINQ to XML) (C#)

18/03/2020 • 7 minutes to read • Edit Online

Modificar un árbol XML directamente es un enfoque tradicional para cambiar la forma de un documento XML. Una aplicación típica carga un documento en un almacén de datos como DOM o LINQ to XML; utiliza una interfaz de programación para insertar nodos, eliminar nodos o cambiar el contenido de los nodos y, a continuación, guarda el XML en un archivo o lo transmite a través de una red.

LINQ to XML permite otro enfoque que es útil en muchos escenarios: la *construcción funcional*. La construcción funcional trata la modificación de datos como un problema de transformación en lugar de una manipulación detallada de un almacén de datos. Si puede tomar una representación de datos y transformarla eficientemente de una a otra, el resultado es el mismo que si ha tomado un almacén de datos y lo ha manipulado de alguna manera para que tome otra forma. Un aspecto fundamental del enfoque de construcción funcional es pasar los resultados de las consultas a los constructores XDocument y XElement.

En muchos casos puede escribir el código de transformación en una fracción del tiempo que tardaría en manipular el almacén de datos y ese código es más eficaz y fácil de mantener. En esos casos, aunque el enfoque de transformación puede necesitar más potencia de procesamiento, es una forma más efectiva de modificar datos. Si un desarrollador está familiarizado con el enfoque funcional, el código resultante es en muchos casos más fácil de entender. Resulta sencillo buscar el código que modifica cada parte del árbol.

El enfoque en el que se modifica un árbol XML directamente es mucho más familiar para muchos programadores de DOM, mientras que el código escrito usando el enfoque funcional puede ser poco familiar para un desarrollador que aún no comprende este enfoque. Si solo tiene que realizar una pequeña modificación en un árbol XML grande, el enfoque en el que se modifica un árbol directamente en muchos casos consumirá menos tiempo de CPU.

Este tema proporciona un ejemplo que se implementa con ambos enfoques.

Transformar atributos en elementos

Para este ejemplo, supongamos que desea modificar el siguiente documento XML sencillo para que los atributos se conviertan en elementos. Este tema primero presenta el enfoque tradicional de modificación directa. Después muestra el enfoque de construcción funcional.

```
<?xml version="1.0" encoding="utf-8" ?>
<Root Data1="123" Data2="456">
  <Child1>Content</Child1>
</Root>
```

Modificar el árbol XML

Puede escribir código de procedimientos para crear elementos a partir de atributos y después eliminar los atributos de la siguiente manera:

```
XElement root = XElement.Load("Data.xml");
foreach (XAttribute att in root.Attributes()) {
    root.Add(new XElement(att.Name, (string)att));
}
root.Attributes().Remove();
Console.WriteLine(root);
```

Este código genera el siguiente resultado:

```
<Root>
  <Child1>Content</Child1>
  <Data1>123</Data1>
  <Data2>456</Data2>
</Root>
```

Enfoque de construcción funcional

Por el contrario, un enfoque funcional consta de un código para formar un nuevo árbol, eligiendo y seleccionando elementos y atributos del árbol de origen y transformándolos, según convenga, a medida que se agregan al nuevo árbol. El enfoque funcional tiene el siguiente aspecto:

```
XElement root = XElement.Load("Data.xml");
XElement newTree = new XElement("Root",
    root.Element("Child1"),
    from att in root.Attributes()
    select new XElement(att.Name, (string)att)
);
Console.WriteLine(newTree);
```

Este ejemplo produce el mismo XML que el primer ejemplo. No obstante, tenga en cuenta que puede ver realmente la estructura resultante del nuevo XML en el enfoque funcional. Puede ver la creación del elemento `Root`, el código que extrae el elemento `Child1` del árbol de origen y el código que transforma los atributos del árbol de origen a los elementos del nuevo árbol.

El ejemplo funcional de este caso no es más corto que el primer ejemplo y en realidad no es más sencillo. No obstante, si tiene muchos cambios que realizar a un árbol XML, el enfoque no funcional se hará más complejo y difícil de entender. Por el contrario, cuando se usa el enfoque funcional, se sigue formando el XML deseado, incrustando consultas y expresiones según convenga, para extraer el contenido deseado. El enfoque funcional produce código que es más fácil de mantener.

Tenga en cuenta que el enfoque funcional probablemente no tendrá un rendimiento tan bueno como la manipulación del árbol. El principal problema es que el enfoque funcional crea objetos de corta duración. No obstante, la compensación es efectiva si el uso del enfoque funcional permite una mayor productividad del programador.

Éste es un ejemplo muy sencillo, pero sirve para mostrar la diferencia de filosofía entre los dos enfoques. El enfoque funcional ofrece mayores ganancias de productividad para transformar documentos XML de gran tamaño.

Agregar elementos, atributos y nodos a un árbol XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

Puede agregar contenidos (elementos, atributos, comentarios, instrucciones de procesamiento, texto y bloques CDATA) a un árbol XML existente.

Métodos para agregar contenidos

Los métodos siguientes agregan contenidos secundarios a un XElement o a un XDocument:

MÉTODO	DESCRIPTION
Add	Agrega un contenido al final de los contenidos secundarios del XContainer.
AddFirst	Agrega un contenido al comienzo de los contenidos secundarios del XContainer.

Los métodos siguientes agregan contenidos como nodos relacionados de un XNode. El nodo al que se agregan habitualmente contenidos relacionados es XElement, aunque es posible agregar contenidos relacionados válidos a otros tipos de nodos, como por ejemplo, al nodo XText o al nodo XComment.

MÉTODO	DESCRIPTION
AddAfterSelf	Añade un contenido detrás de XNode.
AddBeforeSelf	Agrega contenido antes de XNode.

Ejemplo

Description

El siguiente ejemplo crear dos árboles XML y, a continuación, modifica uno de ellos.

Código

```
XElement srcTree = new XElement("Root",
    new XElement("Element1", 1),
    new XElement("Element2", 2),
    new XElement("Element3", 3),
    new XElement("Element4", 4),
    new XElement("Element5", 5)
);
XElement xmlTree = new XElement("Root",
    new XElement("Child1", 1),
    new XElement("Child2", 2),
    new XElement("Child3", 3),
    new XElement("Child4", 4),
    new XElement("Child5", 5)
);
xmlTree.Add(new XElement("NewChild", "new content"));
xmlTree.Add(
    from el in srcTree.Elements()
    where (int)el > 3
    select el
);
// Even though Child9 does not exist in srcTree, the following statement will not
// throw an exception, and nothing will be added to xmlTree.
xmlTree.Add(srcTree.Element("Child9"));
Console.WriteLine(xmlTree);
```

Comentarios

Este código genera el siguiente resultado:

```
<Root>
  <Child1>1</Child1>
  <Child2>2</Child2>
  <Child3>3</Child3>
  <Child4>4</Child4>
  <Child5>5</Child5>
  <NewChild>new content</NewChild>
  <Element4>4</Element4>
  <Element5>5</Element5>
</Root>
```

Modificar elementos, atributos y nodos de un árbol XML

18/03/2020 • 2 minutes to read • Edit Online

La tabla siguiente resume los métodos y las propiedades que puede usar para modificar un elemento, sus elementos secundarios o sus atributos.

Los métodos siguientes modifican un elemento XElement.

MÉTODO	DESCRIPTION
XElement.Parse	Reemplaza un elemento por XML analizado.
XElement.RemoveAll	Quita todo el contenido (atributos y nodos secundarios) de un elemento.
XElement.RemoveAttributes	Quita los atributos de un elemento.
XElement.ReplaceAll	Reemplaza todo el contenido (nodos secundarios y atributos) de un elemento.
XElement.ReplaceAttributes	Reemplaza los atributos de un elemento.
XElement.SetAttributeValue	Establece el valor de un atributo. Crea el atributo si no existe. Si el valor se establece en `null`, quita el atributo.
XElement.SetElementValue	Establece el valor de un elemento secundario. Crea el elemento si no existe. Si el valor se establece en `null`, quita el elemento.
XElement.Value	Reemplaza el contenido (nodos secundarios) de un elemento por el texto especificado.
XElement.SetValue	Establece el valor de un elemento.

Los métodos siguientes modifican un elemento XAttribute.

MÉTODO	DESCRIPTION
XAttribute.Value	Establece el valor de un atributo.
XAttribute.SetValue	Establece el valor de un atributo.

Los métodos siguientes modifican un elemento XNode (incluyendo XElement o XDocument).

MÉTODO	DESCRIPTION
XNode.ReplaceWith	Reemplaza un nodo por contenido nuevo.

Los métodos siguientes modifican un elemento XContainer (XElement o XDocument).

MÉTODO	DESCRIPTION
XContainer.ReplaceNodes	Reemplaza los nodos secundarios por contenido nuevo.

Quitar elementos, atributos y nodos de un árbol XML (C#)

18/03/2020 • 3 minutes to read • Edit Online

Puede modificar un árbol XML mediante la eliminación de elementos, atributos y otros tipos de nodos.

Quitar un elemento o un atributo de un documento XML resulta sencillo. Sin embargo, al quitar colecciones de elementos o atributos, primero debe materializar una colección en una lista y, a continuación, eliminar los elementos o los atributos de ésta. El mejor método consiste en usar el método de extensión Remove, que se ocupará de todo esto.

El motivo principal radica en que la mayoría de las colecciones que se recuperan de un árbol XML se producen con una ejecución aplazada. Si no las materializa primero en una lista, o bien si no usa los métodos de extensión, es posible que aparezca una clase determinada de errores. Para obtener más información, vea Errores en códigos declarativos/imperativos mixtos (LINQ to XML) (C#).

Los siguientes métodos sirven para quitar nodos y atributos de un árbol XML.

MÉTODO	DESCRIPTION
XAttribute.Remove	Quita un elemento XAttribute de su elemento primario.
XContainer.RemoveNodes	Quita los nodos secundarios de un elemento XContainer de la colección.
XElement.RemoveAll	Quita el contenido y los atributos de un elemento XElement.
XElement.RemoveAttributes	Quita los atributos de un elemento XElement.
XElement.SetAttributeValue	Si pasa `null` para el valor, quita el atributo.
XElement.SetElementValue	Si pasa `null` para el valor, quita el elemento secundario.
XNode.Remove	Quita un elemento XNode de su elemento primario.
Extensions.Remove	Quita todos los atributos o elementos de la colección de origen de su elemento primario.

Ejemplo

Description

Este ejemplo demuestra tres métodos para quitar elementos. Primero, quita un solo elemento. En segundo lugar, recupera una colección de elementos, los materializa con el operador Enumerable.ToList y quita la colección. Por último, recupera una colección de elementos y los quita con el método de extensión Remove.

Para obtener más información sobre el operador ToList, vea Convertir tipos de datos (C#).

Código

```
XElement root = XElement.Parse(@"<Root>
    <Child1>
        <GrandChild1/>
        <GrandChild2/>
        <GrandChild3/>
    </Child1>
    <Child2>
        <GrandChild4/>
        <GrandChild5/>
        <GrandChild6/>
    </Child2>
    <Child3>
        <GrandChild7/>
        <GrandChild8/>
        <GrandChild9/>
    </Child3>
</Root>");
root.Element("Child1").Element("GrandChild1").Remove();
root.Element("Child2").Elements().ToList().Remove();
root.Element("Child3").Elements().Remove();
Console.WriteLine(root);
```

Comentarios

Este código genera el siguiente resultado:

```
<Root>
  <Child1>
    <GrandChild2 />
    <GrandChild3 />
  </Child1>
  <Child2 />
  <Child3 />
</Root>
```

Tenga en cuenta que el primer elemento descendiente del secundario se ha quitado de `Child1`. Todos los elementos descendientes del secundario se han quitado de `Child2` y `Child3`.

Mantener pares nombre/valor (C#)

18/03/2020 • 4 minutes to read • Edit Online

Son muchas las aplicaciones que necesitan mantener información que se almacena mejor en forma de pares de nombre/valor. Esta información podría contener datos sobre configuración o valores globales. LINQ to XML incluye métodos que facilitan la operación de mantener un conjunto de pares nombre/valor. Puede almacenar la información como atributos o como un conjunto de elementos secundarios.

Una diferencia existente entre almacenar la información como atributos o como elementos secundarios es que los atributos tienen, como restricción, que solo puede existir un atributo con un nombre en particular para un elemento. Esto no se aplica a los elementos secundarios.

SetAttributeValue y SetElementValue

Los dos métodos que facilitan el mantenimiento de pares nombre/valor son SetAttributeValue y SetElementValue. La semántica de ambos métodos es muy similar.

SetAttributeValue permite agregar, modificar o eliminar atributos de un elemento.

- Si llama al método SetAttributeValue con el nombre de un atributo que no existe, éste creará un nuevo atributo y lo agregará al elemento especificado.
- Si llama al método SetAttributeValue con el nombre de un atributo ya existente y con un contenido en particular, se sobrescribirán los contenidos del atributo con el contenido especificado.
- Si llama al método SetAttributeValue con el nombre de un atributo ya existente y pasando nulo en el contenido, se eliminará el atributo de su elemento primario.

SetElementValue permite agregar, modificar o eliminar elementos secundarios de un elemento.

- Si llama al método SetElementValue con el nombre de un elemento secundario que no existe, éste creará un nuevo elemento y lo agregará al elemento especificado.
- Si llama al método SetElementValue con el nombre de un elemento ya existente y con un contenido en particular, se sobrescribirán los contenidos del elemento con el contenido especificado.
- Si llama al método SetElementValue con el nombre de un elemento ya existente y pasando nulo en el contenido, se eliminará el elemento de su elemento primario.

Ejemplo

El siguiente ejemplo crea un elemento que no tiene atributos. A continuación, utiliza el método SetAttributeValue para crear y mantener una lista de pares nombre/valor.

```
// Create an element with no content.
XElement root = new XElement("Root");

// Add a number of name/value pairs as attributes.
root.SetAttributeValue("Top", 22);
root.SetAttributeValue("Left", 20);
root.SetAttributeValue("Bottom", 122);
root.SetAttributeValue("Right", 300);
root.SetAttributeValue("DefaultColor", "Color.Red");
Console.WriteLine(root);

// Replace the value of Top.
root.SetAttributeValue("Top", 10);
Console.WriteLine(root);

// Remove DefaultColor.
root.SetAttributeValue("DefaultColor", null);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<Root Top="22" Left="20" Bottom="122" Right="300" DefaultColor="Color.Red" />
<Root Top="10" Left="20" Bottom="122" Right="300" DefaultColor="Color.Red" />
<Root Top="10" Left="20" Bottom="122" Right="300" />
```

Ejemplo

El siguiente ejemplo crea un elemento que no tiene elementos secundarios. A continuación, utiliza el método SetElementValue para crear y mantener una lista de pares nombre/valor.

```
// Create an element with no content.
XElement root = new XElement("Root");

// Add a number of name/value pairs as elements.
root.SetElementValue("Top", 22);
root.SetElementValue("Left", 20);
root.SetElementValue("Bottom", 122);
root.SetElementValue("Right", 300);
root.SetElementValue("DefaultColor", "Color.Red");
Console.WriteLine(root);
Console.WriteLine("----");

// Replace the value of Top.
root.SetElementValue("Top", 10);
Console.WriteLine(root);
Console.WriteLine("----");

// Remove DefaultColor.
root.SetElementValue("DefaultColor", null);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <Top>22</Top>
  <Left>20</Left>
  <Bottom>122</Bottom>
  <Right>300</Right>
  <DefaultColor>Color.Red</DefaultColor>
</Root>
----
<Root>
  <Top>10</Top>
  <Left>20</Left>
  <Bottom>122</Bottom>
  <Right>300</Right>
  <DefaultColor>Color.Red</DefaultColor>
</Root>
----
<Root>
  <Top>10</Top>
  <Left>20</Left>
  <Bottom>122</Bottom>
  <Right>300</Right>
</Root>
```

Vea también

- SetAttributeValue
- SetElementValue
- Modificar árboles XML (LINQ to XML) (C#)

Procedimiento para cambiar el espacio de nombres de todo un árbol XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

En ocasiones, tendrá que cambiar, mediante programación, el espacio de nombres de un elemento o de un atributo. Esto resulta sencillo con LINQ to XML. Es posible modificar la propiedad XElement.Name. No es posible modificar la propiedad XAttribute.Name, pero es posible copiar con facilidad los atributos en un System.Collections.Generic.List<T>, eliminar los atributos existentes y, a continuación, agregar los atributos nuevos que se encuentran en el espacio de nombres deseado.

Para más información, consulte Información general sobre los espacios de nombres (LINQ to XML) (C#).

Ejemplo

En el código siguiente se crean dos árboles XML que no están en ningún espacio de nombres. A continuación, se cambia el espacio de nombres de cada árbol y se combinan ambos árboles en uno sólo.

```
XElement tree1 = new XElement("Data",
    new XElement("Child", "content",
        new XAttribute("MyAttr", "content")
    )
);
XElement tree2 = new XElement("Data",
    new XElement("Child", "content",
        new XAttribute("MyAttr", "content")
    )
);
XNamespace aw = "http://www.adventure-works.com";
XNamespace ad = "http://www.adatum.com";
// change the namespace of every element and attribute in the first tree
foreach (XElement el in tree1.DescendantsAndSelf())
{
    el.Name = aw.GetName(el.Name.LocalName);
    List<XAttribute> atList = el.Attributes().ToList();
    el.Attributes().Remove();
    foreach (XAttribute at in atList)
        el.Add(new XAttribute(aw.GetName(at.Name.LocalName), at.Value));
}
// change the namespace of every element and attribute in the second tree
foreach (XElement el in tree2.DescendantsAndSelf())
{
    el.Name = ad.GetName(el.Name.LocalName);
    List<XAttribute> atList = el.Attributes().ToList();
    el.Attributes().Remove();
    foreach (XAttribute at in atList)
        el.Add(new XAttribute(ad.GetName(at.Name.LocalName), at.Value));
}
// add attribute namespaces so that the tree will be serialized with
// the aw and ad namespace prefixes
tree1.Add(
    new XAttribute(XNamespace.Xmlns + "aw", "http://www.adventure-works.com")
);
tree2.Add(
    new XAttribute(XNamespace.Xmlns + "ad", "http://www.adatum.com")
);
// create a new composite tree
XElement root = new XElement("Root",
    tree1,
    tree2
);
Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <aw:Data xmlns:aw="http://www.adventure-works.com">
    <aw:Child aw:MyAttr="content">content</aw:Child>
  </aw:Data>
  <ad:Data xmlns:ad="http://www.adatum.com">
    <ad:Child ad:MyAttr="content">content</ad:Child>
  </ad:Data>
</Root>
```

Rendimiento de consultas encadenadas (LINQ to XML) (C#)

18/03/2020 • 3 minutes to read • Edit Online

Una de las ventajas más importantes de LINQ (y LINQ to XML) es que las consultas encadenadas funcionan igual de bien que una sola consulta más grande y complicada.

Una consulta encadenada es una consulta que usa otra consulta como su origen. Por ejemplo, en el siguiente código simple, `query2` es el origen de `query1`:

```
XElement root = new XElement("Root",
    new XElement("Child", 1),
    new XElement("Child", 2),
    new XElement("Child", 3),
    new XElement("Child", 4)
);

var query1 = from x in root.Elements("Child")
             where (int)x >= 3
             select x;

var query2 = from e in query1
             where (int)e % 2 == 0
             select e;

foreach (var i in query2)
    Console.WriteLine("{0}", (int)i);
```

Este ejemplo produce el siguiente resultado:

```
4
```

Esta consulta encadenada proporciona el mismo perfil de rendimiento que si se recorriese en iteración una lista vinculada.

- El eje Elements tiene básicamente el mismo rendimiento que si se recorriese en iteración una lista vinculada. Elements se implementa como iterador con ejecución aplazada. Es decir, realiza trabajo adicional además de recorrer en iteración la lista vinculada, como asignar el objeto iterador y mantener el seguimiento del estado de la ejecución. Este trabajo se puede dividir en dos categorías: el trabajo que se realiza cuando se configura el iterador y el que se lleva a cabo durante cada iteración. El trabajo de configuración es un trabajo mínimo y fijo, mientras que el realizado durante cada iteración es proporcional al número de elementos de la colección de origen.
- En `query1`, la cláusula `where` provoca la llamada al método Where por parte de la consulta. Este método también se implementa como iterador. El trabajo de configuración está formado por la creación de una instancia del delegado que hará referencia a la expresión lambda, más la configuración normal de un iterador. Con cada iteración, se llama al delegado para que ejecute el predicado. El trabajo de configuración y el realizado durante cada iteración es parecido al llevado a cabo mientras se recorre en iteración el eje.
- En `query1`, la cláusula Select provoca la llamada al método Select por parte de la consulta. Este método tiene el mismo perfil de rendimiento que el método Where.
- En `query2`, tanto la cláusula `where` como la cláusula `select` tienen el mismo perfil de rendimiento que en

`query1`.

Por lo tanto, la iteración por `query2` es directamente proporcional al número de elementos del origen de la primera consulta, es decir, tiempo lineal. Un ejemplo de Visual Basic correspondiente tendría el mismo perfil de rendimiento.

Para más información sobre iteradores, vea yield.

Para obtener un tutorial más detallado sobre consultas encadenadas, vea Tutorial: encadenar cadenas juntas.

Objetos XName y XNamespace atomizados (LINQ to XML) (C#)

18/03/2020 • 3 minutes to read • Edit Online

Los objetos XName y XNamespace se *atomizan*; es decir, si contienen el mismo nombre completo, hacen referencia al mismo objeto. De esta forma, se incrementan los beneficios de rendimiento de las consultas: cuando compare la igualdad de dos nombres atomizados, el lenguaje intermedio subyacente solo debe determinar si los dos puntos de referencia apuntan al mismo objeto. El código subyacente no debe hacer comparaciones de cadenas, ya que sería un proceso muy lento.

Semántica de atomización

Atomización significa que si dos objetos XName tienen el mismo nombre local y se encuentran en el mismo espacio de nombres, comparten la misma instancia. De igual forma, si dos objetos XNamespace tienen el mismo URI de espacio de nombres, comparten la misma instancia.

Para que una clase habilite objetos atomizados, el constructor de la clase debe ser privado y no público. La razón es que si el constructor fuera público, podría crea un objeto no atomizado. Las clases XName y XNamespace implementan un operador de conversión implícita para convertir una cadena en XName o XNamespace. De esta forma, obtiene una instancia de estos objetos. No puede obtener ninguna instancia mediante un constructor, ya que no se puede tener acceso a él.

XName y XNamespace también implementan los operadores de igualdad y desigualdad, para determinar si los dos objetos que se comparan son referencias a la misma instancia.

Ejemplo

El siguiente código crea algunos objetos XElement y muestra que nombres idénticos comparten la misma instancia.

```
XElement r1 = new XElement("Root", "data1");
XElement r2 = XElement.Parse("<Root>data2</Root>");

if ((object)r1.Name == (object)r2.Name)
    Console.WriteLine("r1 and r2 have names that refer to the same instance.");
else
    Console.WriteLine("Different");

XName n = "Root";

if ((object)n == (object)r1.Name)
    Console.WriteLine("The name of r1 and the name in 'n' refer to the same instance.");
else
    Console.WriteLine("Different");
```

Este ejemplo produce el siguiente resultado:

```
r1 and r2 have names that refer to the same instance.
The name of r1 and the name in 'n' refer to the same instance.
```

Como se ha mencionado anteriormente, la ventaja de los objetos atomizados es que el usuario utiliza uno de los métodos de eje que toman XName como parámetro, el método de eje solo debe determinar que los dos nombres

hagan referencia a la misma instancia para seleccionar los elementos deseados.

El siguiente ejemplo pasa XName a la llamada del método Descendants, cuyo rendimiento mejora gracias al patrón de atomización.

```
XElement root = new XElement("Root",
    new XElement("C1", 1),
    new XElement("Z1",
        new XElement("C1", 2),
        new XElement("C1", 1)
    )
);

var query = from e in root.Descendants("C1")
            where (int)e == 1
            select e;

foreach (var z in query)
    Console.WriteLine(z);
```

Este ejemplo produce el siguiente resultado:

```
<C1>1</C1>
<C1>1</C1>
```

Atomización previa de objetos XName (LINQ to XML) (C#)

18/03/2020 • 3 minutes to read • Edit Online

Una manera de mejorar el rendimiento en LINQ to XML es realizar una atomización previa de los objetos XName. La atomización previa significa que asigna una cadena a un objeto XName antes de crear el árbol XML utilizando los constructores de las clases XElement y XAttribute. A continuación, en lugar de pasar una cadena al constructor, que utilizaría la conversión implícita de cadena a XName, pasa el objeto XName inicializado.

Esto mejora el rendimiento si crear un árbol XML de gran tamaño en el que se repiten nombres específicos. Para ello, debe declarar e inicializar los objetos XName antes de construir el árbol XML y, a continuación, utilizar los objetos XName en lugar de especificar las cadenas para los nombres de atributo y elemento. Esta técnica puede producir un aumento significativo del rendimiento si va a crear un gran número de elementos (o atributos) con el mismo nombre.

Debería probar la atomización previa con su situación para decidir si debe utilizarla.

Ejemplo

En el siguiente ejemplo se muestra cómo hacerlo.

```
XName Root = "Root";
XName Data = "Data";
XName ID = "ID";

XElement root = new XElement(Root,
    new XElement(Data,
        new XAttribute(ID, "1"),
        "4,100,000"),
    new XElement(Data,
        new XAttribute(ID, "2"),
        "3,700,000"),
    new XElement(Data,
        new XAttribute(ID, "3"),
        "1,150,000")
);

Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <Data ID="1">4,100,000</Data>
  <Data ID="2">3,700,000</Data>
  <Data ID="3">1,150,000</Data>
</Root>
```

En el siguiente ejemplo se muestra la misma técnica donde el documento XML se encuentra en un espacio de nombres:

```
XNamespace aw = "http://www.adventure-works.com";
XName Root = aw + "Root";
XName Data = aw + "Data";
XName ID = "ID";

XElement root = new XElement(Root,
    new XAttribute(XNamespace.Xmlns + "aw", aw),
    new XElement(Data,
        new XAttribute(ID, "1"),
        "4,100,000"),
    new XElement(Data,
        new XAttribute(ID, "2"),
        "3,700,000"),
    new XElement(Data,
        new XAttribute(ID, "3"),
        "1,150,000")
);

Console.WriteLine(root);
```

Este ejemplo produce el siguiente resultado:

```
<aw:Root xmlns:aw="http://www.adventure-works.com">
  <aw:Data ID="1">4,100,000</aw:Data>
  <aw:Data ID="2">3,700,000</aw:Data>
  <aw:Data ID="3">1,150,000</aw:Data>
</aw:Root>
```

El ejemplo siguiente es más similar a lo que probablemente encontrará en el mundo real. En este ejemplo, una consulta proporciona el contenido del elemento:

```
XName Root = "Root";
XName Data = "Data";
XName ID = "ID";

DateTime t1 = DateTime.Now;
XElement root = new XElement(Root,
    from i in System.Linq.Enumerable.Range(1, 100000)
    select new XElement(Data,
        new XAttribute(ID, i),
        i * 5)
);
DateTime t2 = DateTime.Now;

Console.WriteLine("Time to construct:{0}", t2 - t1);
```

El ejemplo anterior se ejecuta mejor que el siguiente, en el que los nombres no se atomizan previamente:

```
DateTime t1 = DateTime.Now;
XElement root = new XElement("Root",
    from i in System.Linq.Enumerable.Range(1, 100000)
    select new XElement("Data",
        new XAttribute("ID", i),
        i * 5)
);
DateTime t2 = DateTime.Now;

Console.WriteLine("Time to construct:{0}", t2 - t1);
```

Vea también

- Objetos XName y XNamespace atomizados (LINQ to XML) (C#)

Consultas compiladas estáticamente (LINQ to XML) (C#)

18/03/2020 • 5 minutes to read • Edit Online

Una de las ventajas de rendimiento más importantes de LINQ to XML, a diferencia de XmlDocument, es que las consultas LINQ to XML se compilan estáticamente, mientras que las consultas XPath deben interpretarse durante la ejecución. Esta característica está incorporada en LINQ to XML, de modo que no tiene que efectuar pasos adicionales para aprovecharla, pero resulta útil comprender la distinción a la hora de elegir entre las dos tecnologías. Este tema explica la diferencia.

Consultas compiladas de forma estática frente a XPath

En el ejemplo siguiente se muestra cómo obtener los elementos descendientes con un nombre especificado y con un atributo con un valor especificado.

Esta es la expresión XPath equivalente: `//Address[@Type='Shipping']`

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

IEnumerable<XElement> list1 =
    from el in po.Descendants("Address")
    where (string)el.Attribute("Type") == "Shipping"
    select el;

foreach (XElement el in list1)
    Console.WriteLine(el);
```

El compilador reescribe la expresión de consulta de este ejemplo con la sintaxis de consulta basada en métodos. En el ejemplo siguiente, que está escrito en la sintaxis de consulta basada en métodos, se producen los mismos resultados que en el anterior:

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

IEnumerable<XElement> list1 =
    po
    .Descendants("Address")
    .Where(el => (string)el.Attribute("Type") == "Shipping");

foreach (XElement el in list1)
    Console.WriteLine(el);
```

El método Where es una extensión del método. Para más información, vea Métodos de extensión. Dado que Where es un método de extensión, la consulta anterior se compila como si estuviera escrita como se muestra a continuación:

```
XDocument po = XDocument.Load("PurchaseOrders.xml");

IEnumerable<XElement> list1 =
    System.Linq.Enumerable.Where(
        po.Descendants("Address"),
        el => (string)el.Attribute("Type") == "Shipping");

foreach (XElement el in list1)
    Console.WriteLine(el);
```

Este ejemplo produce exactamente los mismos resultados que los dos ejemplos anteriores. Esto ilustra el hecho de que las consultas se compilan de forma efectiva en llamadas a método vinculadas estáticamente. Esto, combinado con la semántica de ejecución aplazada de los iteradores, mejora el rendimiento. Para obtener más información sobre la semántica de ejecución aplazada de iteradores, vea Ejecución aplazada y evaluación diferida en LINQ to XML.

> **NOTE**
>
> Estos ejemplos son representativos del código que el compilador podría escribir. La implementación real podría diferir ligeramente de estos ejemplos, pero el rendimiento será el mismo o similar a estos ejemplos.

Ejecutar expresiones XPath con XmlDocument

En el ejemplo de código siguiente se usa XmlDocument para lograr los mismos resultados que en los ejemplos anteriores:

```
XmlReader reader = XmlReader.Create("PurchaseOrders.xml");
XmlDocument doc = new XmlDocument();
doc.Load(reader);
XmlNodeList nl = doc.SelectNodes(".//Address[@Type='Shipping']");
foreach (XmlNode n in nl)
    Console.WriteLine(n.OuterXml);
reader.Close();
```

Esta consulta devuelve el mismo resultado que los ejemplos que usan LINQ to XML; la única diferencia es que LINQ to XML aplica sangría al XML impreso, mientras que XmlDocument no.

No obstante, el enfoque de XmlDocument generalmente no funciona tan bien como LINQ to XML, porque el método SelectNodes debe realizar lo siguiente internamente cada vez que se le llama:

- Analiza la cadena que contiene la expresión XPath, y divide la cadena en tokens.
- Valida los tokens para asegurarse de que la expresión XPath es válida.
- Traduce la expresión a un árbol de expresión interno.
- Recorre en iteración los nodos, y selecciona de forma adecuada los nodos del conjunto de resultados basándose en la evaluación de la expresión.

Esto es bastante más que el trabajo realizado por la consulta LINQ to XML correspondiente. La diferencia de rendimiento específica varía para distintos tipos de consultas, pero en general las consultas LINQ to XML efectúan menos operaciones y, por lo tanto, se ejecutan mejor, que si se evalúan las expresiones XPath con XmlDocument.

Anotaciones de LINQ to XML

18/03/2020 • 2 minutes to read • Edit Online

Las anotaciones de LINQ to XML le permiten asociar cualquier objeto de cualquier tipo con un componente XML de un árbol XML.

Si desea agregar una anotación a un componente XML, como puede ser un XElement o un XAttribute, deberá llamar al método AddAnnotation. Las anotaciones se obtienen por tipos.

Tenga en cuenta que las anotaciones no forman parte del conjunto de información de XML, por lo que no se serializan o deserializan.

Métodos

Puede utilizar los siguientes métodos a la hora de trabajar con anotaciones:

MÉTODO	DESCRIPTION
AddAnnotation	Agrega un objeto a la lista de anotaciones de un XObject.
Annotation	Obtiene el primer objeto de anotación del tipo especificado a partir de un XObject.
Annotations	Obtiene una colección de anotaciones del tipo especificado a partir de un XObject.
RemoveAnnotations	Elimina las anotaciones del tipo especificado de un XObject.

Eventos de LINQ to XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

Los eventos de LINQ to XML permiten recibir notificaciones cuando se modifica un árbol XML.

Puede agregar eventos a una instancia de cualquier XObject. El controlador de eventos recibirá entonces eventos relacionados con las modificaciones realizadas en ese XObject y en cualquiera de sus descendientes. Por ejemplo, puede agregar un controlador de evento a la raíz del árbol y controlar todas las modificaciones que se realicen al árbol desde ese controlador de eventos.

Para obtener ejemplos de eventos LINQ to XML, vea Changing y Changed.

Tipos y eventos

Puede utilizar los siguientes tipos a la hora de trabajar con eventos:

TIPO	DESCRIPTION
XObjectChange	Especifica el tipo de evento cuando se genera éste para un XObject.
XObjectChangeEventArgs	Proporciona datos para los eventos Changing y Changed.

Los siguientes eventos se generan cuando se modifica un árbol XML:

EVENTO	DESCRIPTION
Changing	Se produce justo antes de que este XObject o cualquiera de sus descendientes se vayan a modificar.
Changed	Se produce cuando ha cambiado un XObject o cualquiera de sus descendientes.

Ejemplo

Description

Los eventos resultan útiles cuando desea mantener cierta información de agregado en un árbol XML. Por ejemplo, quizá desee mantener el total de una factura que es la suma de los conceptos de la factura. Este ejemplo utiliza eventos para mantener el total de todos los elementos secundarios que se encuentran bajo el elemento complejo `Items`.

Código

```
XElement root = new XElement("Root",
    new XElement("Total", "0"),
    new XElement("Items")
);
XElement total = root.Element("Total");
XElement items = root.Element("Items");
items.Changed += (object sender, XObjectChangeEventArgs cea) =>
{
    switch (cea.ObjectChange)
    {
        case XObjectChange.Add:
            if (sender is XElement)
                total.Value = ((int)total + (int)(XElement)sender).ToString();
            if (sender is XText)
                total.Value = ((int)total + (int)((XText)sender).Parent).ToString();
            break;
        case XObjectChange.Remove:
            if (sender is XElement)
                total.Value = ((int)total - (int)(XElement)sender).ToString();
            if (sender is XText)
                total.Value = ((int)total - Int32.Parse(((XText)sender).Value)).ToString();
            break;
    }
    Console.WriteLine("Changed {0} {1}",
      sender.GetType().ToString(), cea.ObjectChange.ToString());
};
items.SetElementValue("Item1", 25);
items.SetElementValue("Item2", 50);
items.SetElementValue("Item2", 75);
items.SetElementValue("Item3", 133);
items.SetElementValue("Item1", null);
items.SetElementValue("Item4", 100);
Console.WriteLine("Total:{0}", (int)total);
Console.WriteLine(root);
```

Comentarios

Este código genera el siguiente resultado:

```
Changed System.Xml.Linq.XElement Add
Changed System.Xml.Linq.XElement Add
Changed System.Xml.Linq.XText Remove
Changed System.Xml.Linq.XText Add
Changed System.Xml.Linq.XElement Add
Changed System.Xml.Linq.XElement Remove
Changed System.Xml.Linq.XElement Add
Total:308
<Root>
  <Total>308</Total>
  <Items>
    <Item2>75</Item2>
    <Item3>133</Item3>
    <Item4>100</Item4>
  </Items>
</Root>
```

Programar con nodos (C#)

18/03/2020 • 7 minutes to read • Edit Online

Los desarrolladores de LINQ to XML que deben escribir programar como un editor de XML, un sistema de transformación o un sistema de escritura de informes a menudo deben escribir programas que funcionan en un nivel de granularidad más fino que los elementos y los atributos. A menudo deben trabajar en el nivel del nodo, manipulando nodos de texto, procesando instrucciones y comentarios. En este tema se proporcionan algunos detalles acerca de la programación en el nivel del nodo.

Detalles del nodo

Existen varios detalles de programación que un programador que trabaja en el nivel de nodo debe conocer.

La propiedad primaria de los nodos secundarios de XDocument está establecida en NULL

La propiedad Parent contiene el XElement primario, no el nodo primario. Los nodos secundarios de XDocument no tienen XElement primario. Su elemento primario es el documento, de forma que la propiedad Parent para esos nodos se establece en NULL.

En el siguiente ejemplo se muestra esto:

```
XDocument doc = XDocument.Parse(@"<!-- a comment --><Root/>");
Console.WriteLine(doc.Nodes().OfType<XComment>().First().Parent == null);
Console.WriteLine(doc.Root.Parent == null);
```

Este ejemplo produce el siguiente resultado:

```
True
True
```

Los nodos de texto adyacentes son posibles

En varios modelos de programación XML, los nodos de texto adyacente siempre están combinados. A veces se denomina normalización de los nodos de texto. LINQ to XML no normaliza nodos de texto. Si agrega dos nodos de texto al mismo elemento, tendrá como resultado nodos de texto adyacentes. En cambio, si agrega contenido especificado como una cadena en lugar de un nodo XText, LINQ to XML puede combinar la cadena con un nodo de texto adyacente.

En el siguiente ejemplo se muestra esto:

```
XElement xmlTree = new XElement("Root", "Content");

Console.WriteLine(xmlTree.Nodes().OfType<XText>().Count());

// this does not add a new text node
xmlTree.Add("new content");
Console.WriteLine(xmlTree.Nodes().OfType<XText>().Count());

// this does add a new, adjacent text node
xmlTree.Add(new XText("more text"));
Console.WriteLine(xmlTree.Nodes().OfType<XText>().Count());
```

Este ejemplo produce el siguiente resultado:

```
1
1
2
```

Los nodos de texto adyacentes vacíos son posibles

En algunos modelos de programación XML, se garantiza que los nodos de texto no contienen la cadena vacía. El razonamiento es que ese nodo de texto no tiene ninguna incidencia en la serialización de XML. No obstante, por la misma razón que los nodos de texto adyacentes son posibles, si quita el texto de un nodo de texto estableciendo su valor en la cadena vacía, el nodo de texto en sí no se eliminará.

```
XElement xmlTree = new XElement("Root", "Content");
XText textNode = xmlTree.Nodes().OfType<XText>().First();

// the following line does not cause the removal of the text node.
textNode.Value = "";

XText textNode2 = xmlTree.Nodes().OfType<XText>().First();
Console.WriteLine(">>{0}<<", textNode2);
```

Este ejemplo produce el siguiente resultado:

```
>><<
```

Un nodo de texto vacío incide sobre la serialización

Si un elemento contiene solamente un nodo de texto que está vacío, se serializa con la sintaxis de etiqueta larga: `<Child></Child>`. Si un elemento no contiene ningún nodo secundario, se serializa con la sintaxis de etiqueta corta: `<Child />`.

```
XElement child1 = new XElement("Child1",
    new XText("")
);
XElement child2 = new XElement("Child2");
Console.WriteLine(child1);
Console.WriteLine(child2);
```

Este ejemplo produce el siguiente resultado:

```
<Child1></Child1>
<Child2 />
```

Los espacios de nombres son atributos en el árbol LINQ to XML

Aunque las declaraciones del espacio de nombres tienen una sintaxis idéntica a los atributos, en algunas interfaces de programación como XSLT y XPath, las declaraciones de espacios de nombres no se consideran atributos. No obstante, en LINQ to XML, los espacios de nombres se almacenan como objetos XAttribute en el árbol XML. Si recorre en iteración los atributos de un elemento que contiene una declaración de espacio de nombres, verá la declaración de espacio de nombres como uno de los elementos de la recopilación devuelta.

La propiedad IsNamespaceDeclaration indica si un atributo es una declaración de espacio de nombres.

```
XElement root = XElement.Parse(
@"<Root
    xmlns='http://www.adventure-works.com'
    xmlns:fc='www.fourthcoffee.com'
    AnAttribute='abc'/>");
foreach (XAttribute att in root.Attributes())
    Console.WriteLine("{0}  IsNamespaceDeclaration:{1}", att, att.IsNamespaceDeclaration);
```

Este ejemplo produce el siguiente resultado:

```
xmlns="http://www.adventure-works.com"  IsNamespaceDeclaration:True
xmlns:fc="www.fourthcoffee.com"  IsNamespaceDeclaration:True
AnAttribute="abc"  IsNamespaceDeclaration:False
```

Los métodos del eje XPath no devuelven un espacio en blanco secundario de XDocument

LINQ to XML permite nodos de texto secundarios de un XDocument mientras los nodos de texto contengan solamente espacios en blanco. No obstante, el modelo de objetos de XPath no incluye el espacio en blanco como nodos secundarios de un documento, así que, cuando recorra en iteración los elementos secundarios de XDocument con el eje Nodes, se devolverán los nodos de texto de espacio en blanco. En cambio, cuando recorra en iteración los elementos secundarios de XDocument con los métodos del eje de XPath, no se devolverán los nodos de texto de espacio en blanco.

```
// Create a document with some white-space child nodes of the document.
XDocument root = XDocument.Parse(
@"<?xml version='1.0' encoding='utf-8' standalone='yes'?>

<Root/>

<!--a comment-->
", LoadOptions.PreserveWhitespace);

// count the white-space child nodes using LINQ to XML
Console.WriteLine(root.Nodes().OfType<XText>().Count());

// count the white-space child nodes using XPathEvaluate
Console.WriteLine(((IEnumerable)root.XPathEvaluate("text()")).OfType<XText>().Count());
```

Este ejemplo produce el siguiente resultado:

```
3
0
```

Los objetos XDeclaration no son nodos

Cuando recorra en iteración los nodos secundarios de XDocument, no verá el objeto de declaración XML. Es una propiedad del documento, no un nodo secundario de él.

```
XDocument doc = new XDocument(
    new XDeclaration("1.0", "utf-8", "yes"),
    new XElement("Root")
);
doc.Save("Temp.xml");
Console.WriteLine(File.ReadAllText("Temp.xml"));

// this shows that there is only one child node of the document
Console.WriteLine(doc.Nodes().Count());
```

Este ejemplo produce el siguiente resultado:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Root />
1
```

Errores en códigos declarativos/imperativos mixtos (LINQ to XML) (C#)

18/03/2020 • 9 minutes to read • Edit Online

LINQ to XML incluye numerosos métodos que le permiten modificar directamente un árbol XML. Puede agregar elementos, eliminarlos, cambiar sus contenidos, agregar atributos, etc. Esta interfaz de programación se describe en Modificar árboles XML (LINQ to XML) (C#). Si está llevando a cabo una iteración por uno de los ejes, como puede ser Elements, y está modificando el árbol XML a medida que recorre el eje, es posible que acabe encontrando errores extraños.

En ocasiones, a este problema se le conoce como "El problema de Halloween".

Definición del problema

Cuando se escribe cierto código utilizando LINQ para iterar por una colección, se utiliza un código de estilo declarativo. Es un método que se aproxima más a definir *qué* es lo que quiere, en vez de especificar *cómo* quiere hacerlo. Si escribe un código que 1) obtenga el primer elemento, 2) lo compruebe con una cierta condición, 3) lo modifique y 4) lo coloque nuevamente en la lista, entonces se trata de un código imperativo. Le está indicando al ordenador *cómo* hacer lo que quiere.

Si se combinan ambos estilos de código en una misma operación, es cuando aparecen los problemas. Considere el siguiente caso:

Suponga que tiene una lista vinculada que contiene tres elementos (a, b y c):

```
a -> b -> c
```

Ahora, suponga que desea recorrer la lista vinculada y añadir tres nuevos elementos (a', b' y c'). Desea que la lista vinculada resultante tenga el siguiente aspecto:

```
a -> a' -> b -> b' -> c -> c'
```

De forma que escribe un código que recorre la lista y, que para cada elemento, añade uno nuevo justo después. Lo que ocurrirá es que el código encontrará primero el elemento `a` e insertará `a'` justo después. Ahora, el código pasará al siguiente nodo de la lista, que en este momento será `a'` Entonces agregará un nuevo elemento a la lista, `a''`.

¿Cómo se puede resolver esto en un caso real? Una opción es realizar una copia de la lista vinculada original y crear una lista completamente nueva. O bien, si únicamente está escribiendo un código imperativo, puede encontrar el primer elemento, añadir el nuevo y, a continuación, avanzar dos elementos en la lista, pasando así por encima del elemento recién agregado.

Agregar a medida que se va iterando

Suponga, por ejemplo, que desea escribir un cierto código que, para cada elemento de un árbol, cree un elemento duplicado:

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
foreach (XElement e in root.Elements())
    root.Add(new XElement(e.Name, (string)e));
```

Este código entrará en un bucle infinito. La instrucción `foreach` lleva a cabo una iteración a lo largo del eje `Elements()`, agregando nuevos elementos al elemento `doc`. Al final, acaba realizando dicha iteración por los elementos que se acaban de agregar. Y, dado que coloca los nuevos objetos en cada una de las iteraciones del bucle, llegará un momento en el que consuma toda la memoria disponible.

Puede resolver este problema almacenando en memoria la colección mediante el operador de consulta estándar ToList, de la siguiente forma:

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
foreach (XElement e in root.Elements().ToList())
    root.Add(new XElement(e.Name, (string)e));
Console.WriteLine(root);
```

Ahora el código funciona correctamente. El árbol XML resultante es como sigue:

```
<Root>
  <A>1</A>
  <B>2</B>
  <C>3</C>
  <A>1</A>
  <B>2</B>
  <C>3</C>
</Root>
```

Eliminar a medida que se va iterando

Si desea eliminar todos los nodos que se encuentren en un cierto nivel, podría tener la tentación de escribir un código como el que sigue:

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
foreach (XElement e in root.Elements())
    e.Remove();
Console.WriteLine(root);
```

Sin embargo, no realiza la tarea que deseaba. En esta situación, tras eliminar el primer elemento (A), éste se elimina del árbol XML contenido en la raíz, con lo que el código del método Elements encargado de la iteración no podrá encontrar el siguiente elemento.

Este código anterior genera el siguiente resultado:

```
<Root>
  <B>2</B>
  <C>3</C>
</Root>
```

De nuevo, la solución pasa por llamar a ToList para materializar la colección, de la siguiente forma:

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
foreach (XElement e in root.Elements().ToList())
    e.Remove();
Console.WriteLine(root);
```

Esto genera el siguiente resultado:

```
<Root />
```

Como alternativa, puede eliminar la iteración entera llamando a RemoveAll en el elemento primario:

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
root.RemoveAll();
Console.WriteLine(root);
```

¿Por qué LINQ no puede controlar este comportamiento?

Una posible aproximación sería trasladar todo a memoria en vez de realizar evaluaciones diferidas. No obstante, esto resultaría muy costoso en términos de rendimiento y de utilización de la memoria. De hecho, si LINQ y (LINQ to XML) utilizaran este método, no sería viable en situaciones reales.

Otra posible aproximación consiste en utilizar una cierta sintaxis transaccional en LINQ y hacer que el compilador intente analizar el código para determinar si es necesario materializar alguna colección en particular. Sin embargo, puede resultar extremadamente complejo analizar todo el código que pueda tener efectos secundarios. Observe el código siguiente:

```
var z =
    from e in root.Elements()
    where TestSomeCondition(e)
    select DoMyProjection(e);
```

Un código de análisis así necesitaría analizar los métodos TestSomeCondition y DoMyProjection, así como todos los métodos a los que llaman, con el fin de determinar si existe código con efectos secundarios. Pero no bastaría con que el código de análisis buscara código que tuviera efectos secundarios. Solo debería seleccionar aquel código que tuviera efectos secundarios sobre los elementos secundarios de `root` en este caso en particular.

LINQ to XML no realiza ese tipo de análisis.

Es responsabilidad del programador evitar este tipo de problemas.

Orientación

Primeramente, no mezcle código imperativo con código declarativo.

Incluso en el caso de que conozca con precisión la semántica de las colecciones y de los métodos que modifican el árbol XML y escriba un código que evite este tipo de problemas, éste deberá ser mantenido por otros desarrolladores en un futuro, y es posible que éstos no estén al tanto de esos problemas. Si mezcla estilos de programación declarativa e imperativa, el código resultará más complicado.

Si escribe código que materialice una colección de forma que se eviten todos estos problemas, incluya en él comentarios de forma que los programadores encargados de su mantenimiento puedan comprender la problemática.

En segundo lugar, si el rendimiento y otras consideraciones lo permiten, utilice únicamente código declarativo. No modifique el árbol XML existente. Genere uno nuevo.

```
XElement root = new XElement("Root",
    new XElement("A", "1"),
    new XElement("B", "2"),
    new XElement("C", "3")
);
XElement newRoot = new XElement("Root",
    root.Elements(),
    root.Elements()
);
Console.WriteLine(newRoot);
```

Procedimiento para hacer streaming de fragmentos XML con acceso a la información del encabezado (C#)

18/03/2020 • 6 minutes to read • Edit Online

A veces debe leer arbitrariamente los archivos XML grandes y escribir la aplicación para que la superficie de memoria de esta sea predecible. Si intenta rellenar un árbol XML con un archivo XML de gran tamaño, su utilización de memoria será proporcional al tamaño del archivo (es decir, excesivo). Por consiguiente, debe utilizar en su lugar una técnica de transmisión por secuencias.

Una opción consiste en escribir la aplicación usando XmlReader. Pero quizás prefiera usar LINQ para consultar el árbol XML. En ese caso, puede escribir su propio método de eje personalizado. Para obtener más información, vea Procedimiento para escribir un método de eje de LINQ to XML (C#).

Para escribir su propio método de eje, debe escribir un pequeño método que utiliza los nodos XmlReader para leer hasta que llega a uno de los nodos en el que está interesado. A continuación el método llama a ReadFrom, que lee de XmlReader y crea una instancia de un fragmento XML. A continuación ofrece cada fragmento a través de `yield return` al método que está enumerando su método de eje personalizado. A continuación puede escribir las consultas LINQ en su método de eje personalizado.

Las técnicas de transmisión por secuencias se aplican mejor en situaciones en las que el documento de origen solo se debe procesar una vez y se pueden procesar los elementos en el orden del documento. Ciertos operadores de consulta estándar, como OrderBy, recorren en iteración su origen, recaban todos los datos, los ordenan y finalmente producen el primer elemento de la secuencia. Si utiliza un operador de consulta que materializa su origen antes de producir el primer elemento, no retendrá una superficie de memoria pequeña.

Ejemplo

A veces el problema se pone un poco más interesante. En el siguiente documento XML, el consumidor de su método de eje personalizado también tiene que conocer el nombre del cliente al que pertenece cada elemento.

```
<?xml version="1.0" encoding="utf-8" ?>
<Root>
  <Customer>
    <Name>A. Datum Corporation</Name>
    <Item>
      <Key>0001</Key>
    </Item>
    <Item>
      <Key>0002</Key>
    </Item>
    <Item>
      <Key>0003</Key>
    </Item>
    <Item>
      <Key>0004</Key>
    </Item>
  </Customer>
  <Customer>
    <Name>Fabrikam, Inc.</Name>
    <Item>
      <Key>0005</Key>
    </Item>
    <Item>
      <Key>0006</Key>
    </Item>
    <Item>
      <Key>0007</Key>
    </Item>
    <Item>
      <Key>0008</Key>
    </Item>
  </Customer>
  <Customer>
    <Name>Southridge Video</Name>
    <Item>
      <Key>0009</Key>
    </Item>
    <Item>
      <Key>0010</Key>
    </Item>
  </Customer>
</Root>
```

El enfoque que toma este ejemplo consiste en consultar también esta información de encabezado, guardar la información de encabezado y después crear un pequeño árbol XML que contiene la información de encabezado y el detalle que está enumerando. El método de eje ofrece este nuevo y pequeño árbol XML. La consulta tiene acceso a la información de encabezado así como a la información detallada.

Este enfoque ocupa una pequeña superficie de memoria. Como se ofrecen todos los fragmentos XML detallados, no se guardan referencias al fragmento anterior y está disponible para la recolección de elementos no utilizados. Esta técnica crea muchos objetos de vida corta en el montón.

En el siguiente ejemplo se muestra cómo implementar y utilizar un método de eje personalizado que transmite por secuencias fragmentos XML del archivo especificado por la URI. Este eje personalizado se ha escrito para que espere un documento que tiene los elementos `Customer`, `Name` y `Item`, y que esos elementos se ordenarán como en el documento `Source.xml` anterior. Se trata de una implementación simplista. Una implementación más sólida estaría preparada para analizar un documento no válido.

```
static IEnumerable<XElement> StreamCustomerItem(string uri)
{
    using (XmlReader reader = XmlReader.Create(uri))
    {
        XElement name = null;
        XElement item = null;

        reader.MoveToContent();

        // Parse the file, save header information when encountered, and yield the
        // Item XElement objects as they are created.

        // loop through Customer elements
        while (reader.Read())
        {
            if (reader.NodeType == XmlNodeType.Element
                && reader.Name == "Customer")
            {
                // move to Name element
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Element &&
                        reader.Name == "Name")
                    {
                        name = XElement.ReadFrom(reader) as XElement;
                        break;
                    }
                }

                // loop through Item elements
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.EndElement)
                        break;
                    if (reader.NodeType == XmlNodeType.Element
                        && reader.Name == "Item")
                    {
                        item = XElement.ReadFrom(reader) as XElement;
                        if (item != null) {
                            XElement tempRoot = new XElement("Root",
                                new XElement(name)
                            );
                            tempRoot.Add(item);
                            yield return item;
                        }
                    }
                }
            }
        }
    }
}

static void Main(string[] args)
{
    XElement xmlTree = new XElement("Root",
        from el in StreamCustomerItem("Source.xml")
        where (int)el.Element("Key") >= 3 && (int)el.Element("Key") <= 7
        select new XElement("Item",
            new XElement("Customer", (string)el.Parent.Element("Name")),
            new XElement(el.Element("Key"))
        )
    );
    Console.WriteLine(xmlTree);
}
```

Este código genera el siguiente resultado:

```
<Root>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0003</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0004</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0005</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0006</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0007</Key>
  </Item>
</Root>
```

Procedimiento para realizar una transformación de streaming de documentos XML grandes (C#)

18/03/2020 • 7 minutes to read • Edit Online

A veces tiene que transformar los archivos XML grandes y escribir la aplicación para que sea predecible la superficie en memoria de la aplicación. Si intenta rellenar un árbol XML con un archivo XML de gran tamaño, su utilización de memoria será proporcional al tamaño del archivo (es decir, excesivo). Por consiguiente, debe utilizar en su lugar una técnica de transmisión por secuencias.

Las técnicas de transmisión por secuencias se aplican mejor en situaciones en las que el documento de origen solo se debe procesar una vez y se pueden procesar los elementos en el orden del documento. Ciertos operadores de consulta estándar, como OrderBy, recorren en iteración su origen, recaban todos los datos, los ordenan y finalmente producen el primer elemento de la secuencia. Tenga en cuenta que si utiliza un operador de consulta que materializa su origen antes de producir el primer elemento, no retendrá una superficie de memoria pequeña para la aplicación.

Incluso si se usa la técnica descrita en Procedimiento para hacer streaming de fragmentos XML con acceso a la información del encabezado (C#), si se intenta ensamblar un árbol XML que contiene el documento transformado, el uso de memoria será excesivo.

Hay dos grandes enfoques. Un enfoque consiste en utilizar las características de procesamiento aplazada de XStreamingElement. El otro enfoque consiste en crear un XmlWriter y utilizar las capacidades de LINQ to XML para escribir elementos en un XmlWriter. En este tema se muestran ambos enfoques.

Ejemplo

El ejemplo siguiente se basa en el ejemplo de Procedimiento para hacer streaming de fragmentos XML con acceso a la información del encabezado (C#).

Este ejemplo utiliza las funciones de ejecución aplazada de XStreamingElement para transmitir por secuencias el resultado. Este ejemplo puede transformar un documento muy grande a la vez que mantiene una pequeña superficie de memoria.

Observe que se escribe el eje personalizado (`StreamCustomerItem`) específicamente para que espere un documento que tiene los elementos `Customer`, `Name` e `Item`, y que esos elementos se organizarán como en el documento Source.xml siguiente. No obstante, una implementación más sólida estaría preparada para analizar un documento no válido.

A continuación, se muestra el documento de origen, Source.xml:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Root>
  <Customer>
    <Name>A. Datum Corporation</Name>
    <Item>
      <Key>0001</Key>
    </Item>
    <Item>
      <Key>0002</Key>
    </Item>
    <Item>
      <Key>0003</Key>
    </Item>
    <Item>
      <Key>0004</Key>
    </Item>
  </Customer>
  <Customer>
    <Name>Fabrikam, Inc.</Name>
    <Item>
      <Key>0005</Key>
    </Item>
    <Item>
      <Key>0006</Key>
    </Item>
    <Item>
      <Key>0007</Key>
    </Item>
    <Item>
      <Key>0008</Key>
    </Item>
  </Customer>
  <Customer>
    <Name>Southridge Video</Name>
    <Item>
      <Key>0009</Key>
    </Item>
    <Item>
      <Key>0010</Key>
    </Item>
  </Customer>
</Root>
```

```
static IEnumerable<XElement> StreamCustomerItem(string uri)
{
    using (XmlReader reader = XmlReader.Create(uri))
    {
        XElement name = null;
        XElement item = null;

        reader.MoveToContent();

        // Parse the file, save header information when encountered, and yield the
        // Item XElement objects as they are created.

        // loop through Customer elements
        while (reader.Read())
        {
            if (reader.NodeType == XmlNodeType.Element
                && reader.Name == "Customer")
            {
                // move to Name element
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Element &&
                        reader.Name == "Name")
                    {
                        name = XElement.ReadFrom(reader) as XElement;
                        break;
                    }
                }

                // loop through Item elements
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.EndElement)
                        break;
                    if (reader.NodeType == XmlNodeType.Element
                        && reader.Name == "Item")
                    {
                        item = XElement.ReadFrom(reader) as XElement;
                        if (item != null)
                        {
                            XElement tempRoot = new XElement("Root",
                                new XElement(name)
                            );
                            tempRoot.Add(item);
                            yield return item;
                        }
                    }
                }
            }
        }
    }
}

static void Main(string[] args)
{
    XStreamingElement root = new XStreamingElement("Root",
        from el in StreamCustomerItem("Source.xml")
        select new XElement("Item",
            new XElement("Customer", (string)el.Parent.Element("Name")),
            new XElement(el.Element("Key"))
        )
    );
    root.Save("Test.xml");
    Console.WriteLine(File.ReadAllText("Test.xml"));
}
```

Este código genera el siguiente resultado:

```
<?xml version="1.0" encoding="utf-8"?>
<Root>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0001</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0002</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0003</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0004</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0005</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0006</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0007</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0008</Key>
  </Item>
  <Item>
    <Customer>Southridge Video</Customer>
    <Key>0009</Key>
  </Item>
  <Item>
    <Customer>Southridge Video</Customer>
    <Key>0010</Key>
  </Item>
</Root>
```

Ejemplo

El ejemplo siguiente también se basa en el ejemplo de Procedimiento para hacer streaming de fragmentos XML con acceso a la información del encabezado (C#).

Este ejemplo utiliza la capacidad de LINQ to XML para escribir elementos en un XmlWriter. Este ejemplo puede transformar un documento muy grande a la vez que mantiene una pequeña superficie de memoria.

Observe que se escribe el eje personalizado (`StreamCustomerItem`) específicamente para que espere un documento que tiene los elementos `Customer`, `Name` e `Item`, y que esos elementos se organizarán como en el documento Source.xml siguiente. Una implementación más sólida, sin embargo, validaría el documento de origen con XSD o se prepararía para analizar un documento no válido.

Este ejemplo utiliza el mismo documento de origen, Source.xml, que el ejemplo anterior de este tema. También produce exactamente el mismo resultado.

Se prefiere usar la clase XStreamingElement para la transmisión por secuencias de la salida XML a sobrescribir una

clase XmlWriter.

```
static IEnumerable<XElement> StreamCustomerItem(string uri)
{
    using (XmlReader reader = XmlReader.Create(uri))
    {
        XElement name = null;
        XElement item = null;

        reader.MoveToContent();

        // Parse the file, save header information when encountered, and yield the
        // Item XElement objects as they are created.

        // loop through Customer elements
        while (reader.Read())
        {
            if (reader.NodeType == XmlNodeType.Element
                && reader.Name == "Customer")
            {
                // move to Name element
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.Element &&
                        reader.Name == "Name")
                    {
                        name = XElement.ReadFrom(reader) as XElement;
                        break;
                    }
                }

                // loop through Item elements
                while (reader.Read())
                {
                    if (reader.NodeType == XmlNodeType.EndElement)
                        break;
                    if (reader.NodeType == XmlNodeType.Element
                        && reader.Name == "Item")
                    {
                        item = XElement.ReadFrom(reader) as XElement;
                        if (item != null) {
                            XElement tempRoot = new XElement("Root",
                                new XElement(name)
                            );
                            tempRoot.Add(item);
                            yield return item;
                        }
                    }
                }
            }
        }
    }
}

static void Main(string[] args)
{
    IEnumerable<XElement> srcTree =
        from el in StreamCustomerItem("Source.xml")
        select new XElement("Item",
            new XElement("Customer", (string)el.Parent.Element("Name")),
            new XElement(el.Element("Key"))
        );
    XmlWriterSettings xws = new XmlWriterSettings();
    xws.OmitXmlDeclaration = true;
    xws.Indent = true;
    using (XmlWriter xw = XmlWriter.Create("Output.xml", xws)) {
        xw.WriteStartElement("Root");
        foreach (XElement el in srcTree)
```

```
            el.WriteTo(xw);
        xw.WriteEndElement();
    }

    string str = File.ReadAllText("Output.xml");
    Console.WriteLine(str);
}
```

Este código genera el siguiente resultado:

```
<Root>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0001</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0002</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0003</Key>
  </Item>
  <Item>
    <Customer>A. Datum Corporation</Customer>
    <Key>0004</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0005</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0006</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0007</Key>
  </Item>
  <Item>
    <Customer>Fabrikam, Inc.</Customer>
    <Key>0008</Key>
  </Item>
  <Item>
    <Customer>Southridge Video</Customer>
    <Key>0009</Key>
  </Item>
  <Item>
    <Customer>Southridge Video</Customer>
    <Key>0010</Key>
  </Item>
</Root>
```

Procedimiento para leer y escribir un documento codificado (C#)

18/03/2020 • 2 minutes to read • Edit Online

Para crear un documento XML codificado, se agrega un XDeclaration al árbol XML, estableciendo la codificación en el nombre de página de códigos deseado.

Los valores devueltos por WebName son valores válidos.

Si lee un documento codificado, la propiedad Encoding estará establecida en el nombre de la página de códigos.

Si establece Encoding en un nombre de página de códigos válido, LINQ to XML se serializará con la codificación especificada.

Ejemplo

En el ejemplo siguiente se crean dos documentos, uno con codificación utf-8 y otro con codificación utf-16. A continuación se cargan los documentos y se imprime la codificación en la consola.

```
Console.WriteLine("Creating a document with utf-8 encoding");
XDocument encodedDoc8 = new XDocument(
    new XDeclaration("1.0", "utf-8", "yes"),
    new XElement("Root", "Content")
);
encodedDoc8.Save("EncodedUtf8.xml");
Console.WriteLine("Encoding is:{0}", encodedDoc8.Declaration.Encoding);
Console.WriteLine();

Console.WriteLine("Creating a document with utf-16 encoding");
XDocument encodedDoc16 = new XDocument(
    new XDeclaration("1.0", "utf-16", "yes"),
    new XElement("Root", "Content")
);
encodedDoc16.Save("EncodedUtf16.xml");
Console.WriteLine("Encoding is:{0}", encodedDoc16.Declaration.Encoding);
Console.WriteLine();

XDocument newDoc8 = XDocument.Load("EncodedUtf8.xml");
Console.WriteLine("Encoded document:");
Console.WriteLine(File.ReadAllText("EncodedUtf8.xml"));
Console.WriteLine();
Console.WriteLine("Encoding of loaded document is:{0}", newDoc8.Declaration.Encoding);
Console.WriteLine();

XDocument newDoc16 = XDocument.Load("EncodedUtf16.xml");
Console.WriteLine("Encoded document:");
Console.WriteLine(File.ReadAllText("EncodedUtf16.xml"));
Console.WriteLine();
Console.WriteLine("Encoding of loaded document is:{0}", newDoc16.Declaration.Encoding);
```

Este ejemplo produce el siguiente resultado:

```
Creating a document with utf-8 encoding
Encoding is:utf-8

Creating a document with utf-16 encoding
Encoding is:utf-16

Encoded document:
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Root>Content</Root>

Encoding of loaded document is:utf-8

Encoded document:
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<Root>Content</Root>

Encoding of loaded document is:utf-16
```

Vea también

- XDeclaration.Encoding

Usar XSLT para transformar un árbol XML (C#)

18/03/2020 • 2 minutes to read • Edit Online

Puede crear un árbol XML, crear un objeto XmlReader desde el árbol XML, crear un nuevo documento y crear un objeto XmlWriter que escribirá en el nuevo documento. A continuación, puede invocar la transformación XSLT y pasar XmlReader y XmlWriter a la transformación. Después de que se complete correctamente la transformación, se rellenará el nuevo árbol XML con los resultados de la transformación.

Ejemplo

```
string xslt = @"<?xml version='1.0'?>
    <xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'>
        <xsl:template match='/Parent'>
            <Root>
                <C1>
                <xsl:value-of select='Child1'/>
                </C1>
                <C2>
                <xsl:value-of select='Child2'/>
                </C2>
            </Root>
        </xsl:template>
    </xsl:stylesheet>";

var oldDocument = new XDocument(
    new XElement("Parent",
        new XElement("Child1", "Child1 data"),
        new XElement("Child2", "Child2 data")
    )
);

var newDocument = new XDocument();

using (var stringReader = new StringReader(xslt))
{
    using (XmlReader xsltReader = XmlReader.Create(stringReader))
    {
        var transformer = new XslCompiledTransform();
        transformer.Load(xsltReader);
        using (XmlReader oldDocumentReader = oldDocument.CreateReader())
        {
            using (XmlWriter newDocumentWriter = newDocument.CreateWriter())
            {
                transformer.Transform(oldDocumentReader, newDocumentWriter);
            }
        }
    }
}

string result = newDocument.ToString();
Console.WriteLine(result);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <C1>Child1 data</C1>
  <C2>Child2 data</C2>
</Root>
```

Vea también

- XContainer.CreateWriter
- XNode.CreateReader

Procedimiento para usar anotaciones para transformar árboles LINQ to XML en un estilo XSLT (C#)

18/03/2020 • 10 minutes to read • Edit Online

Se puede usar anotaciones para facilitar transformaciones de un árbol XML.

Algunos documentos XML están "basados en el documento con contenido mixto". Con estos documentos no se conoce necesariamente la forma de los nodos secundarios de un elemento. Por ejemplo, un nodo que contiene texto puede tener el siguiente aspecto:

```
<text>A phrase with <b>bold</b> and <i>italic</i> text.</text>
```

Para cualquier nodo de texto dado puede haber cualquier número de elementos secundarios `<b>` y `<i>`. Este enfoque se extiende a otras situaciones diferentes, por ejemplo páginas que contienen diversos elementos secundarios, como párrafos normales, párrafos con viñetas o mapas de bits. Las celdas de una tabla pueden contener texto, listas desplegables o mapas de bits. Una de las características principal del XML centrado en documentos es que no se sabe qué elemento secundario puede tener cualquier elemento particular.

Si desea transformar elementos de un árbol en el que no conoce el elemento secundario de los elementos que desea transformar, entonces este enfoque que usa anotaciones en un enfoque efectivo.

El resumen del enfoque es:

- En primer lugar, anote elementos en el árbol con un elemento de sustitución.
- En segundo lugar, recorra en iteración todo el árbol, creando un nuevo árbol en el que se reemplaza cada elemento son su anotación. Este ejemplo implementa la iteración y la creación de un nuevo árbol en una función con el nombre `XForm`.

En detalle, el enfoque consiste en:

- Ejecutar una o más consultas LINQ to XML que devuelvan el conjunto de elementos que desea transformar de una forma a otra. Para cada elemento de la consulta, agregue un nuevo objeto XElement como anotación al elemento. Este nuevo elemento sustituirá el elemento anotado en el nuevo árbol transformado. Es un código sencillo de escribir, tal como lo demuestra el ejemplo.
- El nuevo elemento que se agrega como anotación puede contener nodos secundarios; puede formar un subárbol con cualquier forma que se desee.
- Existe una regla especial; si un nodo secundario del nuevo elemento está en un espacio de nombres diferente, un espacio de nombres creado con esa finalidad (en este ejemplo, el espacio de nombres es `http://www.microsoft.com/LinqToXmlTransform/2007`), entonces el elemento secundario no se copia al nuevo árbol. En su lugar, si el espacio de nombres es el espacio de nombres especial mencionado anteriormente y el nombre local del elemento es `ApplyTransforms`, los nodos secundarios del elemento del árbol de origen se recorren en iteración y se copian al nuevo árbol (con la excepción que los elementos secundarios anotados se transforman de acuerdo con esas reglas).
- Esto es parecido a la especificación de transformaciones en XSL. La consulta que selecciona un conjunto de nodos es análoga a la expresión XPath para una plantilla. El código para crear el nuevo XElement que se

guarda como anotación es análogo al constructor de secuencias en XSL y el elemento `ApplyTransforms` es análogo en función al elemento `xsl:apply-templates` en XSL.

- Una ventaja de este enfoque es que al formular consultas siempre se escriben consultas en el árbol de origen sin modificar. No hay que preocuparse de cómo afectan a las consultas que se están escribiendo las modificaciones del árbol.

Transformar un árbol

El primer ejemplo cambia el nombre de todos los nodos `Paragraph` a `para`.

```
XNamespace xf = "http://www.microsoft.com/LinqToXmlTransform/2007";
XName at = xf + "ApplyTransforms";

XElement root = XElement.Parse(@"
<Root>
    <Paragraph>This is a sentence with <b>bold</b> and <i>italic</i> text.</Paragraph>
    <Paragraph>More text.</Paragraph>
</Root>");

// replace Paragraph with para
foreach (var el in root.Descendants("Paragraph"))
    el.AddAnnotation(
        new XElement("para",
            // same idea as xsl:apply-templates
            new XElement(xf + "ApplyTransforms")
        )
    );

// The XForm method, shown later in this topic, accomplishes the transform
XElement newRoot = XForm(root);

Console.WriteLine(newRoot);
```

Este ejemplo produce el siguiente resultado:

```
<Root>
  <para>This is a sentence with <b>bold</b> and <i>italic</i> text.</para>
  <para>More text.</para>
</Root>
```

Una transformación más complicada

El siguiente ejemplo consulta el árbol y calcula la media y la suma de los elementos `Data` y los agrega como nuevos elementos al árbol.

```
XNamespace xf = "http://www.microsoft.com/LinqToXmlTransform/2007";
XName at = xf + "ApplyTransforms";

XElement data = new XElement("Root",
    new XElement("Data", 20),
    new XElement("Data", 10),
    new XElement("Data", 3)
);

// while adding annotations, you can query the source tree all you want,
// as the tree is not mutated while annotating.
var avg = data.Elements("Data").Select(z => (Decimal)z).Average();
data.AddAnnotation(
    new XElement("Root",
        new XElement(xf + "ApplyTransforms"),
        new XElement("Average", $"{avg:F4}"),
        new XElement("Sum",
            data
            .Elements("Data")
            .Select(z => (int)z)
            .Sum()
        )
    )
);

Console.WriteLine("Before Transform");
Console.WriteLine("----------------");
Console.WriteLine(data);
Console.WriteLine();
Console.WriteLine();

// The XForm method, shown later in this topic, accomplishes the transform
XElement newData = XForm(data);

Console.WriteLine("After Transform");
Console.WriteLine("----------------");
Console.WriteLine(newData);
```

Este ejemplo produce el siguiente resultado:

```
Before Transform
----------------
<Root>
  <Data>20</Data>
  <Data>10</Data>
  <Data>3</Data>
</Root>

After Transform
----------------
<Root>
  <Data>20</Data>
  <Data>10</Data>
  <Data>3</Data>
  <Average>11.0000</Average>
  <Sum>33</Sum>
</Root>
```

Llevar a cabo la transformación

Una pequeña función, `XForm`, crea un nuevo árbol transformado a partir del árbol original anotado.

- El pseudocódigo para la función es bastante sencillo:

```
The function takes an XElement as an argument and returns an XElement.
If an element has an XElement annotation, then
    Return a new XElement
        The name of the new XElement is the annotation element's name.
        All attributes are copied from the annotation to the new node.
        All child nodes are copied from the annotation, with the
            exception that the special node xf:ApplyTransforms is
            recognized, and the source element's child nodes are
            iterated. If the source child node is not an XElement, it
            is copied to the new tree. If the source child is an
            XElement, then it is transformed by calling this function
            recursively.
If an element is not annotated
    Return a new XElement
        The name of the new XElement is the source element's name
        All attributes are copied from the source element to the
            destination's element.
        All child nodes are copied from the source element.
        If the source child node is not an XElement, it is copied to
            the new tree. If the source child is an XElement, then it
            is transformed by calling this function recursively.
```

A continuación se muestra la implementación de esta función:

```
// Build a transformed XML tree per the annotations
static XElement XForm(XElement source)
{
    XNamespace xf = "http://www.microsoft.com/LinqToXmlTransform/2007";
    XName at = xf + "ApplyTransforms";

    if (source.Annotation<XElement>() != null)
    {
        XElement anno = source.Annotation<XElement>();
        return new XElement(anno.Name,
            anno.Attributes(),
            anno
            .Nodes()
            .Select(
                (XNode n) =>
                {
                    XElement annoEl = n as XElement;
                    if (annoEl != null)
                    {
                        if (annoEl.Name == at)
                            return (object)(
                                source.Nodes()
                                .Select(
                                    (XNode n2) =>
                                    {
                                        XElement e2 = n2 as XElement;
                                        if (e2 == null)
                                            return n2;
                                        else
                                            return XForm(e2);
                                    }
                                )
                            );
                        else
                            return n;
                    }
                    else
                        return n;
                }
            )
        );
    }
    else
    {
        return new XElement(source.Name,
            source.Attributes(),
            source
                .Nodes()
                .Select(n =>
                {
                    XElement el = n as XElement;
                    if (el == null)
                        return n;
                    else
                        return XForm(el);
                }
                )
        );
    }
}
```

Ejemplo completo

El siguiente código es un ejemplo completo que incluye la función `XForm`. Incluye unos pocos usos típicos de este tipo de transformación:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml;
using System.Xml.Linq;

class Program
{
    static XNamespace xf = "http://www.microsoft.com/LinqToXmlTransform/2007";
    static XName at = xf + "ApplyTransforms";

    // Build a transformed XML tree per the annotations
    static XElement XForm(XElement source)
    {
        if (source.Annotation<XElement>() != null)
        {
            XElement anno = source.Annotation<XElement>();
            return new XElement(anno.Name,
                anno.Attributes(),
                anno
                .Nodes()
                .Select(
                    (XNode n) =>
                    {
                        XElement annoEl = n as XElement;
                        if (annoEl != null)
                        {
                            if (annoEl.Name == at)
                                return (object)(
                                    source.Nodes()
                                    .Select(
                                        (XNode n2) =>
                                        {
                                            XElement e2 = n2 as XElement;
                                            if (e2 == null)
                                                return n2;
                                            else
                                                return XForm(e2);
                                        }
                                    )
                                );
                            else
                                return n;
                        }
                        else
                            return n;
                    }
                )
            );
        }
        else
        {
            return new XElement(source.Name,
                source.Attributes(),
                source
                    .Nodes()
                    .Select(n =>
                    {
                        XElement el = n as XElement;
                        if (el == null)
                            return n;
                        else
                            return XForm(el);
                    }
                    )
            );
        }
```

```
    }

    static void Main(string[] args)
    {
        XElement root = new XElement("Root",
            new XComment("A comment"),
            new XAttribute("Att1", 123),
            new XElement("Child", 1),
            new XElement("Child", 2),
            new XElement("Other",
                new XElement("GC", 3),
                new XElement("GC", 4)
            ),
            XElement.Parse(
              "<SomeMixedContent>This is <i>an</i> element that " +
              "<b>has</b> some mixed content</SomeMixedContent>"),
            new XElement("AnUnchangedElement", 42)
        );

        // each of the following serves the same semantic purpose as
        // XSLT templates and sequence constructors

        // replace Child with NewChild
        foreach (var el in root.Elements("Child"))
            el.AddAnnotation(new XElement("NewChild", (string)el));

        // replace first GC with GrandChild, add an attribute
        foreach (var el in root.Descendants("GC").Take(1))
            el.AddAnnotation(
                new XElement("GrandChild",
                    new XAttribute("ANewAttribute", 999),
                    (string)el
                )
            );

        // replace Other with NewOther, add new child elements around original content
        foreach (var el in root.Elements("Other"))
            el.AddAnnotation(
                new XElement("NewOther",
                    new XElement("MyNewChild", 1),
                    // same idea as xsl:apply-templates
                    new XElement(xf + "ApplyTransforms"),
                    new XElement("ChildThatComesAfter")
                )
            );

        // change name of element that has mixed content
        root.Descendants("SomeMixedContent").First().AddAnnotation(
            new XElement("MixedContent",
                new XElement(xf + "ApplyTransforms")
            )
        );

        // replace <b> with <Bold>
        foreach (var el in root.Descendants("b"))
            el.AddAnnotation(
                new XElement("Bold",
                    new XElement(xf + "ApplyTransforms")
                )
            );

        // replace <i> with <Italic>
        foreach (var el in root.Descendants("i"))
            el.AddAnnotation(
                new XElement("Italic",
                    new XElement(xf + "ApplyTransforms")
                )
            );
```

```
        Console.WriteLine("Before Transform");
        Console.WriteLine("----------------");
        Console.WriteLine(root);
        Console.WriteLine();
        Console.WriteLine();
        XElement newRoot = XForm(root);

        Console.WriteLine("After Transform");
        Console.WriteLine("----------------");
        Console.WriteLine(newRoot);
    }
}
```

Este ejemplo produce el siguiente resultado:

```
Before Transform
----------------
<Root Att1="123">
  <!--A comment-->
  <Child>1</Child>
  <Child>2</Child>
  <Other>
    <GC>3</GC>
    <GC>4</GC>
  </Other>
  <SomeMixedContent>This is <i>an</i> element that <b>has</b> some mixed content</SomeMixedContent>
  <AnUnchangedElement>42</AnUnchangedElement>
</Root>

After Transform
----------------
<Root Att1="123">
  <!--A comment-->
  <NewChild>1</NewChild>
  <NewChild>2</NewChild>
  <NewOther>
    <MyNewChild>1</MyNewChild>
    <GrandChild ANewAttribute="999">3</GrandChild>
    <GC>4</GC>
    <ChildThatComesAfter />
  </NewOther>
  <MixedContent>This is <Italic>an</Italic> element that <Bold>has</Bold> some mixed content</MixedContent>
  <AnUnchangedElement>42</AnUnchangedElement>
</Root>
```

Procedimiento para serializar con XmlSerializer (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema presenta un ejemplo que serializa y deserializa información mediante XmlSerializer.

Ejemplo

El siguiente ejemplo crea una serie de objetos que contienen a su vez objetos XElement. A continuación, los serializa en una secuencia de memoria y los deserializa de ésta.

```
using System;
using System.IO;
using System.Linq;
using System.Xml;
using System.Xml.Serialization;
using System.Xml.Linq;

public class XElementContainer
{
    public XElement member;

    public XElementContainer()
    {
        member = XLinqTest.CreateXElement();
    }

    public override string ToString()
    {
        return member.ToString();
    }
}

public class XElementNullContainer
{
    public XElement member;

    public XElementNullContainer()
    {
    }
}

class XLinqTest
{
    static void Main(string[] args)
    {
        Test<XElementNullContainer>(new XElementNullContainer());
        Test<XElement>(CreateXElement());
        Test<XElementContainer>(new XElementContainer());
    }

    public static XElement CreateXElement()
    {
        XNamespace ns = "http://www.adventure-works.com";
        return new XElement(ns + "aw");
    }

    static void Test<T>(T obj)
    {
        using (MemoryStream stream = new MemoryStream())
        {
            XmlSerializer s = new XmlSerializer(typeof(T));
            Console.WriteLine("Testing for type: {0}", typeof(T));
            s.Serialize(XmlWriter.Create(stream), obj);
            stream.Flush();
            stream.Seek(0, SeekOrigin.Begin);
            object o = s.Deserialize(XmlReader.Create(stream));
            Console.WriteLine("  Deserialized type: {0}", o.GetType());
        }
    }
}
```

Este ejemplo produce el siguiente resultado:

```
Testing for type: XElementNullContainer
  Deserialized type: XElementNullContainer
Testing for type: System.Xml.Linq.XElement
  Deserialized type: System.Xml.Linq.XElement
Testing for type: XElementContainer
  Deserialized type: XElementContainer
```

Procedimiento para serializar con DataContractSerializer (C#)

18/03/2020 • 2 minutes to read • Edit Online

Este tema presenta un ejemplo que serializa y deserializa información mediante DataContractSerializer.

Ejemplo

El siguiente ejemplo crea una serie de objetos que contienen a su vez objetos XElement. A continuación, los serializa en archivos de texto y después los deserializa a partir de archivos de texto.

```
using System;
using System.Xml;
using System.Xml.Linq;
using System.IO;
using System.Runtime.Serialization;

public class XLinqTest
{
    public static void Main()
    {
        Test<XElement>(CreateXElement());
        Test<XElementContainer>(new XElementContainer());
        Test<XElementNullContainer>(new XElementNullContainer());
    }

    public static void Test<T>(T obj)
    {
        DataContractSerializer s = new DataContractSerializer(typeof(T));
        using (FileStream fs = File.Open("test" + typeof(T).Name + ".xml", FileMode.Create))
        {
            Console.WriteLine("Testing for type: {0}", typeof(T));
            s.WriteObject(fs, obj);
        }
        using (FileStream fs = File.Open("test" + typeof(T).Name + ".xml", FileMode.Open))
        {
            object s2 = s.ReadObject(fs);
            if (s2 == null)
                Console.WriteLine("  Deserialized object is null (Nothing in VB)");
            else
                Console.WriteLine("  Deserialized type: {0}", s2.GetType());
        }
    }

    public static XElement CreateXElement()
    {
        return new XElement(XName.Get("NameInNamespace", "http://www.adventure-works.org"));
    }
}

[DataContract]
public class XElementContainer
{
    [DataMember]
    public XElement member;

    public XElementContainer()
    {
        member = XLinqTest.CreateXElement();
    }
}

[DataContract]
public class XElementNullContainer
{
    [DataMember]
    public XElement member;

    public XElementNullContainer()
    {
        member = null;
    }
}
```

Este ejemplo produce el siguiente resultado:

```
Testing for type: System.Xml.Linq.XElement
  Deserialized type: System.Xml.Linq.XElement
Testing for type: XElementContainer
  Deserialized type: XElementContainer
Testing for type: XElementNullContainer
  Deserialized type: XElementNullContainer
```

Seguridad de LINQ to XML (C#)

18/03/2020 • 16 minutes to read • Edit Online

Este tema describe los problemas de seguridad asociados a LINQ to XML. Además, proporciona algunas indicaciones para reducir la exposición a esos problemas de seguridad.

Información general de seguridad de LINQ to XML

LINQ to XML se ha diseñado más para la comodidad de programación que para aplicaciones de servidor con estrictos requisitos de seguridad. La mayoría de escenarios XML consisten en el procesamiento de documentos XML de confianza en lugar del procesamiento de documentos XML que no son de confianza que se suben a un servidor. LINQ to XML está optimizado para esos escenarios.

Si debe procesar datos que no son de confianza de orígenes desconocidos, Microsoft recomienda usar una instancia de la clase XmlReader que se ha configurado para filtrar ataques de denegación de servicio (DoS) XML.

Si ha configurado XmlReader para mitigar los ataques de denegación de servicio, puede usar ese lector para rellenar un árbol de LINQ to XML y seguir sacando provecho de las mejoras de productividad del programador de LINQ to XML. Muchas técnicas de mitigación del riesgo implican crear lectores configurados para mitigar el problema de seguridad y crear a continuación instancias de un árbol XML mediante el lector configurado.

XML es intrínsecamente vulnerable a ataques de denegación de servicio porque los documentos no tienen límite en el tamaño, la profundidad, el tamaño de nombre de elemento, etc. Independientemente del componente que use para procesar XML, siempre debería estar preparado para reciclar el dominio de la aplicación si utiliza demasiados recursos.

Reducción del riesgo de ataques de XML, XSD, XPath y XSLT

LINQ to XML se compila sobre XmlReader y XmlWriter. LINQ to XML admite XSD y XPath mediante métodos de extensión en los espacios de nombres System.Xml.Schema y System.Xml.XPath. El uso de las clases XmlReader, XPathNavigator y XmlWriter combinado con LINQ to XML, puede invocar XSLT para transformar árboles XML.

Cuando trabaja en un entorno menos seguro, existen varios problemas de seguridad asociados con XML y con el uso de clases en System.Xml, System.Xml.Schema, System.Xml.XPath y System.Xml.Xsl. Algunos de los problemas son:

- XSD, XPath y XSLT son lenguajes basados en cadenas en los que puede especificar operaciones que consumen mucho tiempo o memoria. Es responsabilidad de los programadores que toman cadenas XSD, XPath o XSLT de orígenes que no son de confianza validar que dichas cadenas no son malintencionadas o supervisar y reducir la posibilidad de que la evaluación de esas cadenas provoque un consumo excesivo de recursos del sistema.
- Los esquemas XSD (incluyendo los esquemas alineadas) son intrínsecamente vulnerables a ataques de denegación de servicio. No se deben aceptar esquemas de orígenes que no sean de confianza.
- XSD y XSLT pueden incluir referencias a otros archivos y esas referencias pueden producir ataques entre dominios y zonas.
- Las entidades externas de DTD pueden tener como resultado ataques entre dominios y zonas.
- Las DTD son vulnerables a ataques de denegación de servicio.
- Los documentos XML con una profundidad excepcional pueden plantear problemas de denegación de

servicio. Es posible que desee limitar la profundidad de documentos XML.

- No se admiten componentes como, por ejemplo, objetos NameTable, XmlNamespaceManager y XmlResolver, de ensamblados que no sean de confianza.
- Lectura de datos en fragmentos para mitigar ataques de documentos de gran tamaño.
- Los bloques de scripts de las hojas de estilos XSLT pueden dar lugar a varios ataques.
- Validar cuidadosamente antes de crear expresiones XPath dinámicas.

Problemas de seguridad de LINQ to XML

Los problemas de seguridad de este tema no se presentan en ningún orden específico. Todos los problemas son importantes y deben solucionarse de forma adecuada.

Un ataque de elevación de privilegios que tiene éxito proporciona a un ensamblado malintencionado más control sobre su entorno. Un ataque de elevación de privilegios que tiene éxito puede tener como resultado una revelación de datos, una denegación de servicio, etc.

Las aplicaciones no deben revelar datos a los usuarios que no están autorizados a verlos.

Los ataques de denegación de servicio hacen que el analizador XML o LINQ to XML consuman excesiva memoria o tiempo de CPU. Los ataques de denegación de servicio se consideran menos graves que los ataques de elevación de privilegios o los ataques de revelación de datos. No obstante, son importantes en un escenario en el que un servidor debe procesar documentos XML de orígenes que no sean de confianza.

Las excepciones y los mensajes de error pueden revelar datos

La descripción de un error puede revelar datos como la transformación de datos, los nombres de archivos o los detalles de implementación. Los mensajes de error no se deben mostrar a los autores de la llamada que no son de confianza. Debe detectar todos los errores e notificarlos con sus propios mensajes de error.

No llamar a CodeAccessPermissions.Assert en un controlador de eventos

Un ensamblado puede tener más o menos permisos. Un ensamblado con más permisos cuenta con un mayor control sobre el equipo y sus entornos.

Si el código de un ensamblado con más permisos llama a CodeAccessPermission.Assert en un controlador de eventos y, a continuación, el árbol XML se pasa a un ensamblado malintencionado que tiene permisos restringidos, el ensamblado malintencionado puede hacer que se genere un evento. Como el evento ejecuta código que está en el ensamblado con más permisos, el ensamblado malintencionado funcionará con privilegios elevados.

Microsoft recomienda no llamar nunca a CodeAccessPermission.Assert en un controlador de eventos.

Las DTD no son seguras

Las entidades de DTD son inseguras intrínsecamente. Un documento XML malintencionado que contiene DTD puede hacer que el analizador use toda la memoria y todo el tiempo de CPU, lo que provocaría un ataque de denegación de servicio. Por lo tanto, en LINQ to XML, el procesamiento de DTD está desactivado de forma predeterminada. No debe aceptar DTD de orígenes que no sean de confianza.

Un ejemplo de DTD aceptadas de orígenes que no son de confianza es una aplicación web que permite a los usuarios cargar un archivo XML que hace referencia a una DTD y un archivo de DTD. Al validar el archivo, una DTD malintencionada puede ejecutar un ataque de denegación de servicio en el servidor. Otro ejemplo de DTD aceptadas de orígenes que no son de confianza es hacer referencia a una DTD en un recurso compartido de red que también permite el acceso FTP anónimo.

Evitar la asignación excesiva de búfer

Los desarrolladores de aplicaciones deben tener en cuenta que orígenes de datos extremadamente grandes pueden provocar un agotamiento de los recursos y ataques de denegación de servicio.

Si un usuario malintencionado envía o carga un documento XML de gran tamaño, puede hacer que LINQ to XML consuma demasiados recursos del sistema. Esto puede representar un ataque de denegación de servicio. Para evitarlo, puede ajustar la propiedad XmlReaderSettings.MaxCharactersInDocument y crear un lector con un límite en el tamaño del documento que puede cargar. El lector se usa a continuación para crear el árbol XML.

Por ejemplo, si sabe que el tamaño máximo previsto de los documentos XML que provienen de un origen que no es de confianza va a ser inferior a 50 KB, establezca XmlReaderSettings.MaxCharactersInDocument en 100.000. Esto no será un inconveniente para el procesamiento de sus documentos XML y a la vez contrarrestará amenazas de denegación de servicio en las que se podrían cargar documentos que consumirían grandes cantidades de memoria.

Evitar la expansión excesiva de entidades

Uno de los ataques de denegación de servicio conocidos cuando se usa una DTD es un documento que provoca una expansión excesiva de entidades. Para evitar que esto se produzca, puede establecer la propiedad XmlReaderSettings.MaxCharactersFromEntities y crear un lector que tenga un límite en el número de caracteres producidos por la expansión de la entidad. El lector se usa a continuación para crear el árbol XML.

Limitar la profundidad de la jerarquía XML

Un posible ataque de denegación de servicio consiste en enviar un documento que tiene una profundidad de jerarquía excesiva. Para evitarlo, puede encapsular XmlReader en su propia clase que cuenta la profundidad de los elementos. Si la profundidad supera un nivel razonable predeterminado, puede finalizar el procesamiento del documento malintencionado.

Protegerse frente a implementaciones de XmlReader o XmlWriter que no son de confianza

Los administradores deben comprobar que las implementaciones de XmlReader o XmlWriter proporcionadas de forma externa tienen nombres seguros y se han registrado en la configuración del equipo. Esto evita que se cargue un código malintencionado que simula ser un lector o escritor.

Liberar periódicamente objetos que hacen referencia a XName

Para protegerse frente a cierto tipo de ataques, los programadores de aplicaciones deben liberar todos los objetos que hacen referencia a un objeto XName en el dominio de la aplicación de forma periódica.

Protegerse frente a nombres XML aleatorios

Las aplicaciones que toman datos de orígenes que no son de confianza deben plantearse el uso de XmlReader, que se incluye en el código personalizado para inspeccionar la posibilidad de nombres y espacios de nombres XML aleatorios. Si se detectan nombres y espacios de nombres XML aleatorios, la aplicación puede finalizar el procesamiento del documento malintencionado.

Es posible que desee limitar los nombres incluidos en cualquier espacio de nombres dado (incluyendo nombres que no están en ningún espacio de nombres) a una cantidad razonable.

Los componentes de software que comparten un árbol de LINQ to XML pueden tener acceso a las anotaciones

LINQ to XML se puede usar para compilar canalizaciones de procesamiento en las que diferentes componentes de la aplicación cargan, validan, consultan, transforman, actualizan y guardan datos XML que se pasan entre componentes como árboles XML. Esto puede ayudarle a optimizar el rendimiento, porque la sobrecarga de carga y serialización de objetos a texto XML se realiza únicamente en los extremos de la canalización. Sin embargo, los desarrolladores deben tener en cuenta que otros componentes puedes tener acceso a todas las anotaciones y controladores de eventos creados por un componente. Esto puede crear varias vulnerabilidades cuando los componentes tienen diferentes niveles de confianza. Para compilar canalizaciones seguras en componentes con un nivel de confianza inferior, debe serializar los objetos LINQ to XML a texto XML antes de pasar los datos a un componente que no es de confianza.

Common Language Runtime (CLR) proporciona cierta seguridad. Por ejemplo, un componente que no incluye una clase privada no puede tener acceso a anotaciones con claves creadas por esa clase. No obstante, las anotaciones pueden ser eliminadas por componentes que no las leen. Esto se podría usar como ataque de manipulación.

Vea también

- Guía de programación (LINQ to XML) (C#)

Archivo XML de ejemplo: Pedido de compra común (LINQ to XML)

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo es un pedido de compra típico.

PurchaseOrder.xml

```
<?xml version="1.0"?>
<PurchaseOrder PurchaseOrderNumber="99503" OrderDate="1999-10-20">
  <Address Type="Shipping">
    <Name>Ellen Adams</Name>
    <Street>123 Maple Street</Street>
    <City>Mill Valley</City>
    <State>CA</State>
    <Zip>10999</Zip>
    <Country>USA</Country>
  </Address>
  <Address Type="Billing">
    <Name>Tai Yee</Name>
    <Street>8 Oak Avenue</Street>
    <City>Old Town</City>
    <State>PA</State>
    <Zip>95819</Zip>
    <Country>USA</Country>
  </Address>
  <DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
  <Items>
    <Item PartNumber="872-AA">
      <ProductName>Lawnmower</ProductName>
      <Quantity>1</Quantity>
      <USPrice>148.95</USPrice>
      <Comment>Confirm this is electric</Comment>
    </Item>
    <Item PartNumber="926-AA">
      <ProductName>Baby Monitor</ProductName>
      <Quantity>2</Quantity>
      <USPrice>39.98</USPrice>
      <ShipDate>1999-05-21</ShipDate>
    </Item>
  </Items>
</PurchaseOrder>
```

Archivo XML de ejemplo: Pedido de compra común en un espacio de nombres

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo es un pedido de compra típico. El XML se encuentra en un espacio de nombres.

PurchaseOrderInNamespace.xml

```
<?xml version="1.0"?>
<aw:PurchaseOrder
    aw:PurchaseOrderNumber="99503"
    aw:OrderDate="1999-10-20"
    xmlns:aw="http://www.adventure-works.com">
  <aw:Address aw:Type="Shipping">
    <aw:Name>Ellen Adams</aw:Name>
    <aw:Street>123 Maple Street</aw:Street>
    <aw:City>Mill Valley</aw:City>
    <aw:State>CA</aw:State>
    <aw:Zip>10999</aw:Zip>
    <aw:Country>USA</aw:Country>
  </aw:Address>
  <aw:Address aw:Type="Billing">
    <aw:Name>Tai Yee</aw:Name>
    <aw:Street>8 Oak Avenue</aw:Street>
    <aw:City>Old Town</aw:City>
    <aw:State>PA</aw:State>
    <aw:Zip>95819</aw:Zip>
    <aw:Country>USA</aw:Country>
  </aw:Address>
  <aw:DeliveryNotes>Please leave packages in shed by driveway.</aw:DeliveryNotes>
  <aw:Items>
    <aw:Item aw:PartNumber="872-AA">
      <aw:ProductName>Lawnmower</aw:ProductName>
      <aw:Quantity>1</aw:Quantity>
      <aw:USPrice>148.95</aw:USPrice>
      <aw:Comment>Confirm this is electric</aw:Comment>
    </aw:Item>
    <aw:Item aw:PartNumber="926-AA">
      <aw:ProductName>Baby Monitor</aw:ProductName>
      <aw:Quantity>2</aw:Quantity>
      <aw:USPrice>39.98</aw:USPrice>
      <aw:ShipDate>1999-05-21</aw:ShipDate>
    </aw:Item>
  </aw:Items>
</aw:PurchaseOrder>
```

Archivo XML de ejemplo: Varios pedidos de compra (LINQ to XML)

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo contiene varios pedidos de compra.

PurchaseOrders.xml

```
<?xml version="1.0"?>
<PurchaseOrders>
  <PurchaseOrder PurchaseOrderNumber="99503" OrderDate="1999-10-20">
    <Address Type="Shipping">
      <Name>Ellen Adams</Name>
      <Street>123 Maple Street</Street>
      <City>Mill Valley</City>
      <State>CA</State>
      <Zip>10999</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Tai Yee</Name>
      <Street>8 Oak Avenue</Street>
      <City>Old Town</City>
      <State>PA</State>
      <Zip>95819</Zip>
      <Country>USA</Country>
    </Address>
    <DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
    <Items>
      <Item PartNumber="872-AA">
        <ProductName>Lawnmower</ProductName>
        <Quantity>1</Quantity>
        <USPrice>148.95</USPrice>
        <Comment>Confirm this is electric</Comment>
      </Item>
      <Item PartNumber="926-AA">
        <ProductName>Baby Monitor</ProductName>
        <Quantity>2</Quantity>
        <USPrice>39.98</USPrice>
        <ShipDate>1999-05-21</ShipDate>
      </Item>
    </Items>
  </PurchaseOrder>
  <PurchaseOrder PurchaseOrderNumber="99505" OrderDate="1999-10-22">
    <Address Type="Shipping">
      <Name>Cristian Osorio</Name>
      <Street>456 Main Street</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Cristian Osorio</Name>
      <Street>456 Main Street</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
```

```
      <Country>USA</Country>
    </Address>
    <DeliveryNotes>Please notify me before shipping.</DeliveryNotes>
    <Items>
      <Item PartNumber="456-NM">
        <ProductName>Power Supply</ProductName>
        <Quantity>1</Quantity>
        <USPrice>45.99</USPrice>
      </Item>
    </Items>
  </PurchaseOrder>
  <PurchaseOrder PurchaseOrderNumber="99504" OrderDate="1999-10-22">
    <Address Type="Shipping">
      <Name>Jessica Arnold</Name>
      <Street>4055 Madison Ave</Street>
      <City>Seattle</City>
      <State>WA</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Jessica Arnold</Name>
      <Street>4055 Madison Ave</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <Items>
      <Item PartNumber="898-AZ">
        <ProductName>Computer Keyboard</ProductName>
        <Quantity>1</Quantity>
        <USPrice>29.99</USPrice>
      </Item>
      <Item PartNumber="898-AM">
        <ProductName>Wireless Mouse</ProductName>
        <Quantity>1</Quantity>
        <USPrice>14.99</USPrice>
      </Item>
    </Items>
  </PurchaseOrder>
</PurchaseOrders>
```

Archivo XML de ejemplo: Varios pedidos de compra en un espacio de nombres

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo contiene varios pedidos de compra. El XML se encuentra en un espacio de nombres.

PurchaseOrdersInNamespace.xml

```
<?xml version="1.0" encoding="utf-8"?>
<aw:PurchaseOrders xmlns:aw="http://www.adventure-works.com">
  <aw:PurchaseOrder aw:PurchaseOrderNumber="99503" aw:OrderDate="1999-10-20">
    <aw:Address aw:Type="Shipping">
      <aw:Name>Ellen Adams</aw:Name>
      <aw:Street>123 Maple Street</aw:Street>
      <aw:City>Mill Valley</aw:City>
      <aw:State>CA</aw:State>
      <aw:Zip>10999</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:Address aw:Type="Billing">
      <aw:Name>Tai Yee</aw:Name>
      <aw:Street>8 Oak Avenue</aw:Street>
      <aw:City>Old Town</aw:City>
      <aw:State>PA</aw:State>
      <aw:Zip>95819</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:DeliveryNotes>Please leave packages in shed by driveway.</aw:DeliveryNotes>
    <aw:Items>
      <aw:Item aw:PartNumber="872-AA">
        <aw:ProductName>Lawnmower</aw:ProductName>
        <aw:Quantity>1</aw:Quantity>
        <aw:USPrice>148.95</aw:USPrice>
        <aw:Comment>Confirm this is electric</aw:Comment>
      </aw:Item>
      <aw:Item aw:PartNumber="926-AA">
        <aw:ProductName>Baby Monitor</aw:ProductName>
        <aw:Quantity>2</aw:Quantity>
        <aw:USPrice>39.98</aw:USPrice>
        <aw:ShipDate>1999-05-21</aw:ShipDate>
      </aw:Item>
    </aw:Items>
  </aw:PurchaseOrder>
  <aw:PurchaseOrder aw:PurchaseOrderNumber="99505" aw:OrderDate="1999-10-22">
    <aw:Address aw:Type="Shipping">
      <aw:Name>Cristian Osorio</aw:Name>
      <aw:Street>456 Main Street</aw:Street>
      <aw:City>Buffalo</aw:City>
      <aw:State>NY</aw:State>
      <aw:Zip>98112</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:Address aw:Type="Billing">
      <aw:Name>Cristian Osorio</aw:Name>
      <aw:Street>456 Main Street</aw:Street>
      <aw:City>Buffalo</aw:City>
      <aw:State>NY</aw:State>
      <aw:Zip>98112</aw:Zip>
      <aw:Country>USA</aw:Country>
```

```
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:DeliveryNotes>Please notify me before shipping.</aw:DeliveryNotes>
    <aw:Items>
      <aw:Item aw:PartNumber="456-NM">
        <aw:ProductName>Power Supply</aw:ProductName>
        <aw:Quantity>1</aw:Quantity>
        <aw:USPrice>45.99</aw:USPrice>
      </aw:Item>
    </aw:Items>
  </aw:PurchaseOrder>
  <aw:PurchaseOrder aw:PurchaseOrderNumber="99504" aw:OrderDate="1999-10-22">
    <aw:Address aw:Type="Shipping">
      <aw:Name>Jessica Arnold</aw:Name>
      <aw:Street>4055 Madison Ave</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98112</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:Address aw:Type="Billing">
      <aw:Name>Jessica Arnold</aw:Name>
      <aw:Street>4055 Madison Ave</aw:Street>
      <aw:City>Buffalo</aw:City>
      <aw:State>NY</aw:State>
      <aw:Zip>98112</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:Address>
    <aw:Items>
      <aw:Item aw:PartNumber="898-AZ">
        <aw:ProductName>Computer Keyboard</aw:ProductName>
        <aw:Quantity>1</aw:Quantity>
        <aw:USPrice>29.99</aw:USPrice>
      </aw:Item>
      <aw:Item aw:PartNumber="898-AM">
        <aw:ProductName>Wireless Mouse</aw:ProductName>
        <aw:Quantity>1</aw:Quantity>
        <aw:USPrice>14.99</aw:USPrice>
      </aw:Item>
    </aw:Items>
  </aw:PurchaseOrder>
</aw:PurchaseOrders>
```

Archivo XML de ejemplo: Configuración de prueba (LINQ to XML)

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Se trata de un archivo de configuración de pruebas.

TestConfig.xml

```
<?xml version="1.0"?>
<Tests>
  <Test TestId="0001" TestType="CMD">
    <Name>Convert number to string</Name>
    <CommandLine>Examp1.EXE</CommandLine>
    <Input>1</Input>
    <Output>One</Output>
  </Test>
  <Test TestId="0002" TestType="CMD">
    <Name>Find succeeding characters</Name>
    <CommandLine>Examp2.EXE</CommandLine>
    <Input>abc</Input>
    <Output>def</Output>
  </Test>
  <Test TestId="0003" TestType="GUI">
    <Name>Convert multiple numbers to strings</Name>
    <CommandLine>Examp2.EXE /Verbose</CommandLine>
    <Input>123</Input>
    <Output>One Two Three</Output>
  </Test>
  <Test TestId="0004" TestType="GUI">
    <Name>Find correlated key</Name>
    <CommandLine>Examp3.EXE</CommandLine>
    <Input>a1</Input>
    <Output>b1</Output>
  </Test>
  <Test TestId="0005" TestType="GUI">
    <Name>Count characters</Name>
    <CommandLine>FinalExamp.EXE</CommandLine>
    <Input>This is a test</Input>
    <Output>14</Output>
  </Test>
  <Test TestId="0006" TestType="GUI">
    <Name>Another Test</Name>
    <CommandLine>Examp2.EXE</CommandLine>
    <Input>Test Input</Input>
    <Output>10</Output>
  </Test>
</Tests>
```

Archivo XML de ejemplo: Configuración de prueba en un espacio de nombres

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Se trata de un archivo de configuración de pruebas. El XML se encuentra en un espacio de nombres.

TestConfigInNamespace.xml

```
<?xml version="1.0"?>
<Tests xmlns="http://www.adatum.com">
  <Test TestId="0001" TestType="CMD">
    <Name>Convert number to string</Name>
    <CommandLine>Examp1.EXE</CommandLine>
    <Input>1</Input>
    <Output>One</Output>
  </Test>
  <Test TestId="0002" TestType="CMD">
    <Name>Find succeeding characters</Name>
    <CommandLine>Examp2.EXE</CommandLine>
    <Input>abc</Input>
    <Output>def</Output>
  </Test>
  <Test TestId="0003" TestType="GUI">
    <Name>Convert multiple numbers to strings</Name>
    <CommandLine>Examp2.EXE /Verbose</CommandLine>
    <Input>123</Input>
    <Output>One Two Three</Output>
  </Test>
  <Test TestId="0004" TestType="GUI">
    <Name>Find correlated key</Name>
    <CommandLine>Examp3.EXE</CommandLine>
    <Input>a1</Input>
    <Output>b1</Output>
  </Test>
  <Test TestId="0005" TestType="GUI">
    <Name>Count characters</Name>
    <CommandLine>FinalExamp.EXE</CommandLine>
    <Input>This is a test</Input>
    <Output>14</Output>
  </Test>
  <Test TestId="0006" TestType="GUI">
    <Name>Another Test</Name>
    <CommandLine>Examp2.EXE</CommandLine>
    <Input>Test Input</Input>
    <Output>10</Output>
  </Test>
</Tests>
```

Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML)

18/03/2020 • 4 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo contiene clientes y pedidos.

En el tema Sample XSD File: Customers and Orders (Archivo XSD de ejemplo: Clientes y pedidos) se incluye un XSD que se puede usar para validar este documento. Usa las características `xs:key` y `xs:keyref` de XSD para establecer que el atributo `CustomerID` del elemento `Customer` es una clave, y para establecer una relación entre el elemento `CustomerID` de cada elemento `Order` y el atributo `CustomerID` de cada elemento `Customer`.

Para conocer un ejemplo de escritura de consultas LINQ que aprovechan esta relación mediante la cláusula `Join`, consulte Procedimiento para combinar dos colecciones (LINQ to XML) (C#).

CustomersOrders.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Root>
  <Customers>
    <Customer CustomerID="GREAL">
      <CompanyName>Great Lakes Food Market</CompanyName>
      <ContactName>Howard Snyder</ContactName>
      <ContactTitle>Marketing Manager</ContactTitle>
      <Phone>(503) 555-7555</Phone>
      <FullAddress>
        <Address>2732 Baker Blvd.</Address>
        <City>Eugene</City>
        <Region>OR</Region>
        <PostalCode>97403</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="HUNGC">
      <CompanyName>Hungry Coyote Import Store</CompanyName>
      <ContactName>Yoshi Latimer</ContactName>
      <ContactTitle>Sales Representative</ContactTitle>
      <Phone>(503) 555-6874</Phone>
      <Fax>(503) 555-2376</Fax>
      <FullAddress>
        <Address>City Center Plaza 516 Main St.</Address>
        <City>Elgin</City>
        <Region>OR</Region>
        <PostalCode>97827</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="LAZYK">
      <CompanyName>Lazy K Kountry Store</CompanyName>
      <ContactName>John Steel</ContactName>
      <ContactTitle>Marketing Manager</ContactTitle>
      <Phone>(509) 555-7969</Phone>
      <Fax>(509) 555-6221</Fax>
      <FullAddress>
        <Address>12 Orchestra Terrace</Address>
        <City>Walla Walla</City>
        <Region>WA</Region>
```

```
        <Region>WA</Region>
        <PostalCode>99362</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="LETSS">
      <CompanyName>Let's Stop N Shop</CompanyName>
      <ContactName>Jaime Yorres</ContactName>
      <ContactTitle>Owner</ContactTitle>
      <Phone>(415) 555-5938</Phone>
      <FullAddress>
        <Address>87 Polk St. Suite 5</Address>
        <City>San Francisco</City>
        <Region>CA</Region>
        <PostalCode>94117</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
  </Customers>
  <Orders>
    <Order>
      <CustomerID>GREAL</CustomerID>
      <EmployeeID>6</EmployeeID>
      <OrderDate>1997-05-06T00:00:00</OrderDate>
      <RequiredDate>1997-05-20T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-05-09T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>3.35</Freight>
        <ShipName>Great Lakes Food Market</ShipName>
        <ShipAddress>2732 Baker Blvd.</ShipAddress>
        <ShipCity>Eugene</ShipCity>
        <ShipRegion>OR</ShipRegion>
        <ShipPostalCode>97403</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>GREAL</CustomerID>
      <EmployeeID>8</EmployeeID>
      <OrderDate>1997-07-04T00:00:00</OrderDate>
      <RequiredDate>1997-08-01T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-07-14T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>4.42</Freight>
        <ShipName>Great Lakes Food Market</ShipName>
        <ShipAddress>2732 Baker Blvd.</ShipAddress>
        <ShipCity>Eugene</ShipCity>
        <ShipRegion>OR</ShipRegion>
        <ShipPostalCode>97403</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>GREAL</CustomerID>
      <EmployeeID>1</EmployeeID>
      <OrderDate>1997-07-31T00:00:00</OrderDate>
      <RequiredDate>1997-08-28T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-08-05T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>116.53</Freight>
        <ShipName>Great Lakes Food Market</ShipName>
        <ShipAddress>2732 Baker Blvd.</ShipAddress>
        <ShipCity>Eugene</ShipCity>
        <ShipRegion>OR</ShipRegion>
        <ShipPostalCode>97403</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>GREAL</CustomerID>
```

```
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1997-07-31T00:00:00</OrderDate>
    <RequiredDate>1997-08-28T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-08-04T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>18.53</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>6</EmployeeID>
    <OrderDate>1997-09-04T00:00:00</OrderDate>
    <RequiredDate>1997-10-02T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-09-10T00:00:00">
      <ShipVia>1</ShipVia>
      <Freight>57.15</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1997-09-25T00:00:00</OrderDate>
    <RequiredDate>1997-10-23T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-09-30T00:00:00">
      <ShipVia>3</ShipVia>
      <Freight>76.13</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1998-01-06T00:00:00</OrderDate>
    <RequiredDate>1998-02-03T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1998-02-04T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>719.78</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1998-03-09T00:00:00</OrderDate>
    <RequiredDate>1998-04-06T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1998-03-18T00:00:00">
```

```
      <ShipVia>2</ShipVia>
      <Freight>33.68</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1998-04-07T00:00:00</OrderDate>
    <RequiredDate>1998-05-05T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1998-04-15T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>25.19</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1998-04-22T00:00:00</OrderDate>
    <RequiredDate>1998-05-20T00:00:00</RequiredDate>
    <ShipInfo>
      <ShipVia>3</ShipVia>
      <Freight>18.84</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1998-04-30T00:00:00</OrderDate>
    <RequiredDate>1998-06-11T00:00:00</RequiredDate>
    <ShipInfo>
      <ShipVia>3</ShipVia>
      <Freight>14.01</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1996-12-06T00:00:00</OrderDate>
    <RequiredDate>1997-01-03T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1996-12-09T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>20.12</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
```

```
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1996-12-25T00:00:00</OrderDate>
    <RequiredDate>1997-01-22T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-01-03T00:00:00">
      <ShipVia>3</ShipVia>
      <Freight>30.34</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1997-01-15T00:00:00</OrderDate>
    <RequiredDate>1997-02-12T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-01-24T00:00:00">
      <ShipVia>1</ShipVia>
      <Freight>0.2</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1997-07-16T00:00:00</OrderDate>
    <RequiredDate>1997-08-13T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-07-21T00:00:00">
      <ShipVia>1</ShipVia>
      <Freight>45.13</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-09-08T00:00:00</OrderDate>
    <RequiredDate>1997-10-06T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-10-15T00:00:00">
      <ShipVia>1</ShipVia>
      <Freight>111.29</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
```

```
    <Order>
      <CustomerID>LAZYK</CustomerID>
      <EmployeeID>1</EmployeeID>
      <OrderDate>1997-03-21T00:00:00</OrderDate>
      <RequiredDate>1997-04-18T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-04-10T00:00:00">
        <ShipVia>3</ShipVia>
        <Freight>7.48</Freight>
        <ShipName>Lazy K Kountry Store</ShipName>
        <ShipAddress>12 Orchestra Terrace</ShipAddress>
        <ShipCity>Walla Walla</ShipCity>
        <ShipRegion>WA</ShipRegion>
        <ShipPostalCode>99362</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>LAZYK</CustomerID>
      <EmployeeID>8</EmployeeID>
      <OrderDate>1997-05-22T00:00:00</OrderDate>
      <RequiredDate>1997-06-19T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-06-26T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>11.92</Freight>
        <ShipName>Lazy K Kountry Store</ShipName>
        <ShipAddress>12 Orchestra Terrace</ShipAddress>
        <ShipCity>Walla Walla</ShipCity>
        <ShipRegion>WA</ShipRegion>
        <ShipPostalCode>99362</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>LETSS</CustomerID>
      <EmployeeID>1</EmployeeID>
      <OrderDate>1997-06-25T00:00:00</OrderDate>
      <RequiredDate>1997-07-23T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-07-04T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>13.73</Freight>
        <ShipName>Let's Stop N Shop</ShipName>
        <ShipAddress>87 Polk St. Suite 5</ShipAddress>
        <ShipCity>San Francisco</ShipCity>
        <ShipRegion>CA</ShipRegion>
        <ShipPostalCode>94117</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>LETSS</CustomerID>
      <EmployeeID>8</EmployeeID>
      <OrderDate>1997-10-27T00:00:00</OrderDate>
      <RequiredDate>1997-11-24T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-11-05T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>51.44</Freight>
        <ShipName>Let's Stop N Shop</ShipName>
        <ShipAddress>87 Polk St. Suite 5</ShipAddress>
        <ShipCity>San Francisco</ShipCity>
        <ShipRegion>CA</ShipRegion>
        <ShipPostalCode>94117</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>LETSS</CustomerID>
      <EmployeeID>6</EmployeeID>
      <OrderDate>1997-11-10T00:00:00</OrderDate>
      <RequiredDate>1997-12-08T00:00:00</RequiredDate>
```

```
      <ShipInfo ShippedDate="1997-11-21T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>45.97</Freight>
        <ShipName>Let's Stop N Shop</ShipName>
        <ShipAddress>87 Polk St. Suite 5</ShipAddress>
        <ShipCity>San Francisco</ShipCity>
        <ShipRegion>CA</ShipRegion>
        <ShipPostalCode>94117</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>LETSS</CustomerID>
      <EmployeeID>4</EmployeeID>
      <OrderDate>1998-02-12T00:00:00</OrderDate>
      <RequiredDate>1998-03-12T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1998-02-13T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>90.97</Freight>
        <ShipName>Let's Stop N Shop</ShipName>
        <ShipAddress>87 Polk St. Suite 5</ShipAddress>
        <ShipCity>San Francisco</ShipCity>
        <ShipRegion>CA</ShipRegion>
        <ShipPostalCode>94117</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
  </Orders>
</Root>
```

Archivo XSD de ejemplo: Clientes y pedidos

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XSD se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo contiene una definición de esquema del Archivo XML de ejemplo: Clientes y pedidos (LINQ to XML). El esquema usa las características `xs:key` y `xs:keyref` de XSD para establecer que el atributo `CustomerID` del elemento `Customer` es una clave, y para establecer una relación entre el elemento `CustomerID` de cada elemento `Order` y el atributo `CustomerID` de cada elemento `Customer`.

Para conocer un ejemplo de escritura de consultas LINQ que aprovechan esta relación mediante la cláusula `Join`, consulte Procedimiento para combinar dos colecciones (LINQ to XML) (C#).

CustomersOrders.xsd

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name='Root'>
    <xs:complexType>
      <xs:sequence>
        <xs:element name='Customers'>
          <xs:complexType>
            <xs:sequence>
              <xs:element name='Customer' type='CustomerType' minOccurs='0' maxOccurs='unbounded' />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name='Orders'>
          <xs:complexType>
            <xs:sequence>
              <xs:element name='Order' type='OrderType' minOccurs='0' maxOccurs='unbounded' />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
    <xs:key name='CustomerIDKey'>
      <xs:selector xpath='Customers/Customer'/>
      <xs:field xpath='@CustomerID'/>
    </xs:key>
    <xs:keyref name='CustomerIDKeyRef' refer='CustomerIDKey'>
      <xs:selector xpath='Orders/Order'/>
      <xs:field xpath='CustomerID'/>
    </xs:keyref>
  </xs:element>
  <xs:complexType name='CustomerType'>
    <xs:sequence>
      <xs:element name='CompanyName' type='xs:string'/>
      <xs:element name='ContactName' type='xs:string'/>
      <xs:element name='ContactTitle' type='xs:string'/>
      <xs:element name='Phone' type='xs:string'/>
      <xs:element name='Fax' minOccurs='0' type='xs:string'/>
      <xs:element name='FullAddress' type='AddressType'/>
    </xs:sequence>
    <xs:attribute name='CustomerID' type='xs:token'/>
  </xs:complexType>
  <xs:complexType name='AddressType'>
    <xs:sequence>
      <xs:element name='Address' type='xs:string'/>
      <xs:element name='City' type='xs:string'/>
```

```
      <xs:element name='Region' type='xs:string'/>
      <xs:element name='PostalCode' type='xs:string' />
      <xs:element name='Country' type='xs:string'/>
    </xs:sequence>
    <xs:attribute name='CustomerID' type='xs:token'/>
  </xs:complexType>
  <xs:complexType name='OrderType'>
    <xs:sequence>
      <xs:element name='CustomerID' type='xs:token'/>
      <xs:element name='EmployeeID' type='xs:token'/>
      <xs:element name='OrderDate' type='xs:dateTime'/>
      <xs:element name='RequiredDate' type='xs:dateTime'/>
      <xs:element name='ShipInfo' type='ShipInfoType'/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name='ShipInfoType'>
    <xs:sequence>
      <xs:element name='ShipVia' type='xs:integer'/>
      <xs:element name='Freight' type='xs:decimal'/>
      <xs:element name='ShipName' type='xs:string'/>
      <xs:element name='ShipAddress' type='xs:string'/>
      <xs:element name='ShipCity' type='xs:string'/>
      <xs:element name='ShipRegion' type='xs:string'/>
      <xs:element name='ShipPostalCode' type='xs:string'/>
      <xs:element name='ShipCountry' type='xs:string'/>
    </xs:sequence>
    <xs:attribute name='ShippedDate' type='xs:dateTime'/>
  </xs:complexType>
</xs:schema>
```

Archivo XML de ejemplo: Clientes y pedidos en un espacio de nombres

18/03/2020 • 3 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo contiene clientes y pedidos. El XML se encuentra en un espacio de nombres.

CustomersOrdersInNamespace.xml

```
<?xml version="1.0" encoding="utf-8"?>
<Root xmlns="http://www.adventure-works.com">
  <Customers>
    <Customer CustomerID="GREAL">
      <CompanyName>Great Lakes Food Market</CompanyName>
      <ContactName>Howard Snyder</ContactName>
      <ContactTitle>Marketing Manager</ContactTitle>
      <Phone>(503) 555-7555</Phone>
      <FullAddress>
        <Address>2732 Baker Blvd.</Address>
        <City>Eugene</City>
        <Region>OR</Region>
        <PostalCode>97403</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="HUNGC">
      <CompanyName>Hungry Coyote Import Store</CompanyName>
      <ContactName>Yoshi Latimer</ContactName>
      <ContactTitle>Sales Representative</ContactTitle>
      <Phone>(503) 555-6874</Phone>
      <Fax>(503) 555-2376</Fax>
      <FullAddress>
        <Address>City Center Plaza 516 Main St.</Address>
        <City>Elgin</City>
        <Region>OR</Region>
        <PostalCode>97827</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="LAZYK">
      <CompanyName>Lazy K Kountry Store</CompanyName>
      <ContactName>John Steel</ContactName>
      <ContactTitle>Marketing Manager</ContactTitle>
      <Phone>(509) 555-7969</Phone>
      <Fax>(509) 555-6221</Fax>
      <FullAddress>
        <Address>12 Orchestra Terrace</Address>
        <City>Walla Walla</City>
        <Region>WA</Region>
        <PostalCode>99362</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
    <Customer CustomerID="LETSS">
      <CompanyName>Let's Stop N Shop</CompanyName>
      <ContactName>Jaime Yorres</ContactName>
      <ContactTitle>Owner</ContactTitle>
      <Phone>(415) 555-5938</Phone>
      <FullAddress>
        <Address>87 Polk St., Suite 5</Address>
```

```
        <Address>87 Polk St. Suite 5</Address>
        <City>San Francisco</City>
        <Region>CA</Region>
        <PostalCode>94117</PostalCode>
        <Country>USA</Country>
      </FullAddress>
    </Customer>
  </Customers>
  <Orders>
    <Order>
      <CustomerID>GREAL</CustomerID>
      <EmployeeID>6</EmployeeID>
      <OrderDate>1997-05-06T00:00:00</OrderDate>
      <RequiredDate>1997-05-20T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-05-09T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>3.35</Freight>
        <ShipName>Great Lakes Food Market</ShipName>
        <ShipAddress>2732 Baker Blvd.</ShipAddress>
        <ShipCity>Eugene</ShipCity>
        <ShipRegion>OR</ShipRegion>
        <ShipPostalCode>97403</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>GREAL</CustomerID>
      <EmployeeID>8</EmployeeID>
      <OrderDate>1997-07-04T00:00:00</OrderDate>
      <RequiredDate>1997-08-01T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-07-14T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>4.42</Freight>
        <ShipName>Great Lakes Food Market</ShipName>
        <ShipAddress>2732 Baker Blvd.</ShipAddress>
        <ShipCity>Eugene</ShipCity>
        <ShipRegion>OR</ShipRegion>
        <ShipPostalCode>97403</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>GREAL</CustomerID>
      <EmployeeID>1</EmployeeID>
      <OrderDate>1997-07-31T00:00:00</OrderDate>
      <RequiredDate>1997-08-28T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-08-05T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>116.53</Freight>
        <ShipName>Great Lakes Food Market</ShipName>
        <ShipAddress>2732 Baker Blvd.</ShipAddress>
        <ShipCity>Eugene</ShipCity>
        <ShipRegion>OR</ShipRegion>
        <ShipPostalCode>97403</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
    <Order>
      <CustomerID>GREAL</CustomerID>
      <EmployeeID>4</EmployeeID>
      <OrderDate>1997-07-31T00:00:00</OrderDate>
      <RequiredDate>1997-08-28T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1997-08-04T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>18.53</Freight>
        <ShipName>Great Lakes Food Market</ShipName>
        <ShipAddress>2732 Baker Blvd.</ShipAddress>
        <ShipCity>Eugene</ShipCity>
        <ShipRegion>OR</ShipRegion>
        <ShipPostalCode>97403</ShipPostalCode>
```

```
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>6</EmployeeID>
    <OrderDate>1997-09-04T00:00:00</OrderDate>
    <RequiredDate>1997-10-02T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-09-10T00:00:00">
      <ShipVia>1</ShipVia>
      <Freight>57.15</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1997-09-25T00:00:00</OrderDate>
    <RequiredDate>1997-10-23T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-09-30T00:00:00">
      <ShipVia>3</ShipVia>
      <Freight>76.13</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1998-01-06T00:00:00</OrderDate>
    <RequiredDate>1998-02-03T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1998-02-04T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>719.78</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1998-03-09T00:00:00</OrderDate>
    <RequiredDate>1998-04-06T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1998-03-18T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>33.68</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
```

```
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1998-04-07T00:00:00</OrderDate>
    <RequiredDate>1998-05-05T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1998-04-15T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>25.19</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1998-04-22T00:00:00</OrderDate>
    <RequiredDate>1998-05-20T00:00:00</RequiredDate>
    <ShipInfo>
      <ShipVia>3</ShipVia>
      <Freight>18.84</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>GREAL</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1998-04-30T00:00:00</OrderDate>
    <RequiredDate>1998-06-11T00:00:00</RequiredDate>
    <ShipInfo>
      <ShipVia>3</ShipVia>
      <Freight>14.01</Freight>
      <ShipName>Great Lakes Food Market</ShipName>
      <ShipAddress>2732 Baker Blvd.</ShipAddress>
      <ShipCity>Eugene</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97403</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1996-12-06T00:00:00</OrderDate>
    <RequiredDate>1997-01-03T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1996-12-09T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>20.12</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1996-12-25T00:00:00</OrderDate>
    <RequiredDate>1997-01-22T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-01-03T00:00:00">
```

```
      <ShipVia>3</ShipVia>
      <Freight>30.34</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>3</EmployeeID>
    <OrderDate>1997-01-15T00:00:00</OrderDate>
    <RequiredDate>1997-02-12T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-01-24T00:00:00">
      <ShipVia>1</ShipVia>
      <Freight>0.2</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>4</EmployeeID>
    <OrderDate>1997-07-16T00:00:00</OrderDate>
    <RequiredDate>1997-08-13T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-07-21T00:00:00">
      <ShipVia>1</ShipVia>
      <Freight>45.13</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>HUNGC</CustomerID>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-09-08T00:00:00</OrderDate>
    <RequiredDate>1997-10-06T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-10-15T00:00:00">
      <ShipVia>1</ShipVia>
      <Freight>111.29</Freight>
      <ShipName>Hungry Coyote Import Store</ShipName>
      <ShipAddress>City Center Plaza 516 Main St.</ShipAddress>
      <ShipCity>Elgin</ShipCity>
      <ShipRegion>OR</ShipRegion>
      <ShipPostalCode>97827</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>LAZYK</CustomerID>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1997-03-21T00:00:00</OrderDate>
    <RequiredDate>1997-04-18T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-04-10T00:00:00">
      <ShipVia>3</ShipVia>
      <Freight>7.48</Freight>
      <ShipName>Lazy K Kountry Store</ShipName>
      <ShipAddress>12 Orchestra Terrace</ShipAddress>
      <ShipCity>Walla Walla</ShipCity>
```

```
      <ShipRegion>WA</ShipRegion>
      <ShipPostalCode>99362</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>LAZYK</CustomerID>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-05-22T00:00:00</OrderDate>
    <RequiredDate>1997-06-19T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-06-26T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>11.92</Freight>
      <ShipName>Lazy K Kountry Store</ShipName>
      <ShipAddress>12 Orchestra Terrace</ShipAddress>
      <ShipCity>Walla Walla</ShipCity>
      <ShipRegion>WA</ShipRegion>
      <ShipPostalCode>99362</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <EmployeeID>1</EmployeeID>
    <OrderDate>1997-06-25T00:00:00</OrderDate>
    <RequiredDate>1997-07-23T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-07-04T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>13.73</Freight>
      <ShipName>Let's Stop N Shop</ShipName>
      <ShipAddress>87 Polk St. Suite 5</ShipAddress>
      <ShipCity>San Francisco</ShipCity>
      <ShipRegion>CA</ShipRegion>
      <ShipPostalCode>94117</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <EmployeeID>8</EmployeeID>
    <OrderDate>1997-10-27T00:00:00</OrderDate>
    <RequiredDate>1997-11-24T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-11-05T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>51.44</Freight>
      <ShipName>Let's Stop N Shop</ShipName>
      <ShipAddress>87 Polk St. Suite 5</ShipAddress>
      <ShipCity>San Francisco</ShipCity>
      <ShipRegion>CA</ShipRegion>
      <ShipPostalCode>94117</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
  <Order>
    <CustomerID>LETSS</CustomerID>
    <EmployeeID>6</EmployeeID>
    <OrderDate>1997-11-10T00:00:00</OrderDate>
    <RequiredDate>1997-12-08T00:00:00</RequiredDate>
    <ShipInfo ShippedDate="1997-11-21T00:00:00">
      <ShipVia>2</ShipVia>
      <Freight>45.97</Freight>
      <ShipName>Let's Stop N Shop</ShipName>
      <ShipAddress>87 Polk St. Suite 5</ShipAddress>
      <ShipCity>San Francisco</ShipCity>
      <ShipRegion>CA</ShipRegion>
      <ShipPostalCode>94117</ShipPostalCode>
      <ShipCountry>USA</ShipCountry>
    </ShipInfo>
  </Order>
```

```
    <Order>
      <CustomerID>LETSS</CustomerID>
      <EmployeeID>4</EmployeeID>
      <OrderDate>1998-02-12T00:00:00</OrderDate>
      <RequiredDate>1998-03-12T00:00:00</RequiredDate>
      <ShipInfo ShippedDate="1998-02-13T00:00:00">
        <ShipVia>2</ShipVia>
        <Freight>90.97</Freight>
        <ShipName>Let's Stop N Shop</ShipName>
        <ShipAddress>87 Polk St. Suite 5</ShipAddress>
        <ShipCity>San Francisco</ShipCity>
        <ShipRegion>CA</ShipRegion>
        <ShipPostalCode>94117</ShipPostalCode>
        <ShipCountry>USA</ShipCountry>
      </ShipInfo>
    </Order>
  </Orders>
</Root>
```

Archivo XML de ejemplo: Datos numéricos (LINQ to XML)

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo contiene datos numéricos para realizar sumas, promedios y agrupaciones.

Data.xml

```
<Root>
  <TaxRate>7.25</TaxRate>
  <Data>
    <Category>A</Category>
    <Quantity>3</Quantity>
    <Price>24.50</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>1</Quantity>
    <Price>89.99</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>5</Quantity>
    <Price>4.95</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>3</Quantity>
    <Price>66.00</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>10</Quantity>
    <Price>.99</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>15</Quantity>
    <Price>29.00</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>8</Quantity>
    <Price>6.99</Price>
  </Data>
</Root>
```

Archivo XML de ejemplo: Datos numéricos de un espacio de nombres

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo contiene datos numéricos para realizar sumas, promedios y agrupaciones. El XML se encuentra en un espacio de nombres.

data

```
<Root xmlns='http://www.adatum.com'>
  <TaxRate>7.25</TaxRate>
  <Data>
    <Category>A</Category>
    <Quantity>3</Quantity>
    <Price>24.50</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>1</Quantity>
    <Price>89.99</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>5</Quantity>
    <Price>4.95</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>3</Quantity>
    <Price>66.00</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>10</Quantity>
    <Price>.99</Price>
  </Data>
  <Data>
    <Category>A</Category>
    <Quantity>15</Quantity>
    <Price>29.00</Price>
  </Data>
  <Data>
    <Category>B</Category>
    <Quantity>8</Quantity>
    <Price>6.99</Price>
  </Data>
</Root>
```

Archivo XML de ejemplo: Libros (LINQ to XML)

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. El archivo contiene información acerca de libros.

books.xml

```
<?xml version="1.0"?>
<Catalog>
   <Book id="bk101">
      <Author>Garghentini, Davide</Author>
      <Title>XML Developer's Guide</Title>
      <Genre>Computer</Genre>
      <Price>44.95</Price>
      <PublishDate>2000-10-01</PublishDate>
      <Description>An in-depth look at creating applications
      with XML.</Description>
   </Book>
   <Book id="bk102">
      <Author>Garcia, Debra</Author>
      <Title>Midnight Rain</Title>
      <Genre>Fantasy</Genre>
      <Price>5.95</Price>
      <PublishDate>2000-12-16</PublishDate>
      <Description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</Description>
   </Book>
</Catalog>
```

Archivo XML de ejemplo: Pedidos de compra consolidados

18/03/2020 • 2 minutes to read • Edit Online

El siguiente archivo XML se usa en numerosos ejemplos de la documentación de LINQ to XML. Este archivo es un conjunto de pedidos de compra con distintas formas de varias compañías. Los pedidos de compra de cada compañía se encuentran en espacios de nombres independientes.

ConsolidatedPurchaseOrders.xml

```
<?xml version="1.0"?>
<PurchaseOrders xmlns="www.contoso.com">
  <PurchaseOrder
      PurchaseOrderNumber="99503"
      OrderDate="1999-10-20">
    <Address Type="Shipping">
      <Name>Ellen Adams</Name>
      <Street>123 Maple Street</Street>
      <City>Mill Valley</City>
      <State>CA</State>
      <Zip>10999</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Tai Yee</Name>
      <Street>8 Oak Avenue</Street>
      <City>Old Town</City>
      <State>PA</State>
      <Zip>95819</Zip>
      <Country>USA</Country>
    </Address>
    <DeliveryNotes>Please leave packages in shed by driveway.</DeliveryNotes>
    <Items>
      <Item PartNumber="872-AA">
        <ProductName>Lawnmower</ProductName>
        <Quantity>1</Quantity>
        <USPrice>148.95</USPrice>
        <Comment>Confirm this is electric</Comment>
      </Item>
      <Item PartNumber="926-AA">
        <ProductName>Baby Monitor</ProductName>
        <Quantity>2</Quantity>
        <USPrice>39.98</USPrice>
        <ShipDate>1999-05-21</ShipDate>
      </Item>
    </Items>
  </PurchaseOrder>
  <PurchaseOrder PurchaseOrderNumber="99505" OrderDate="1999-10-22">
    <Address Type="Shipping">
      <Name>Cristian Osorio</Name>
      <Street>456 Main Street</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Cristian Osorio</Name>
      <Street>456 Main Street</Street>
```

```
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <DeliveryNotes>Please notify by email before shipping.</DeliveryNotes>
    <Items>
      <Item PartNumber="456-NM">
        <ProductName>Power Supply</ProductName>
        <Quantity>1</Quantity>
        <USPrice>45.99</USPrice>
      </Item>
    </Items>
  </PurchaseOrder>
  <PurchaseOrder PurchaseOrderNumber="99504" OrderDate="1999-10-22">
    <Address Type="Shipping">
      <Name>Jessica Arnold</Name>
      <Street>4055 Madison Ave</Street>
      <City>Seattle</City>
      <State>WA</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <Address Type="Billing">
      <Name>Jessica Arnold</Name>
      <Street>4055 Madison Ave</Street>
      <City>Buffalo</City>
      <State>NY</State>
      <Zip>98112</Zip>
      <Country>USA</Country>
    </Address>
    <DeliveryNotes>Please do not deliver on Saturday.</DeliveryNotes>
    <Items>
      <Item PartNumber="898-AZ">
        <ProductName>Computer Keyboard</ProductName>
        <Quantity>1</Quantity>
        <USPrice>29.99</USPrice>
      </Item>
      <Item PartNumber="898-AM">
        <ProductName>Wireless Mouse</ProductName>
        <Quantity>1</Quantity>
        <USPrice>14.99</USPrice>
      </Item>
    </Items>
  </PurchaseOrder>
  <aw:PurchaseOrder
    PONumber="11223"
    Date="2000-01-15"
    xmlns:aw="http://www.adventure-works.com">
    <aw:ShippingAddress>
      <aw:Name>Chris Preston</aw:Name>
      <aw:Street>123 Main St.</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98113</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:ShippingAddress>
    <aw:BillingAddress>
      <aw:Name>Chris Preston</aw:Name>
      <aw:Street>123 Main St.</aw:Street>
      <aw:City>Seattle</aw:City>
      <aw:State>WA</aw:State>
      <aw:Zip>98113</aw:Zip>
      <aw:Country>USA</aw:Country>
    </aw:BillingAddress>
    <aw:DeliveryInstructions>Ship only complete order.</aw:DeliveryInstructions>
    <aw:Item PartNum="LIT-01">
      <aw:ProductID>Litware Networking Card</aw:ProductID>
      <aw:Qty>1</aw:Qty>
```

```
      <aw:Price>20.99</aw:Price>
    </aw:Item>
    <aw:Item PartNum="LIT-25">
      <aw:ProductID>Litware 17in LCD Monitor</aw:ProductID>
      <aw:Qty>1</aw:Qty>
      <aw:Price>199.99</aw:Price>
    </aw:Item>
  </aw:PurchaseOrder>
</PurchaseOrders>
```

Referencia (LINQ to XML)

18/03/2020 • 2 minutes to read • Edit Online

Este tema contiene vínculos a temas de referencia asociados a LINQ to XML.

En esta sección

Si desea obtener documentación de referencia sobre las clases de LINQ to XML, vea System.Xml.Linq.

Si desea obtener documentación de referencia acerca de los métodos de extensión que le permiten validar árboles XML contra un archivo XSD, vea System.Xml.Schema.Extensions.

Si desea obtener documentación de referencia acerca de los métodos de extensión que le permiten evaluar consultas XPath sobre un árbol XML, vea System.Xml.XPath.Extensions.

Vea también

- LINQ to XML (C#)

LINQ to ADO.NET (Página de portal)

18/03/2020 • 3 minutes to read • Edit Online

LINQ to ADO.NET permite consultar sobre cualquier objeto enumerable de ADO.NET mediante el modelo de programación de Language Integrated Query (LINQ).

> **NOTE**
> La documentación de LINQ to ADO.NET se encuentra en la sección ADO.NET del SDK de .NET Framework: LINQ y ADO.NET.

Hay tres tecnologías de Language Integrated Query (LINQ) de ADO.NET independientes: LINQ to DataSet, LINQ to SQL y LINQ to Entities. LINQ to DataSet proporciona consultas más ricas y optimizadas de DataSet, LINQ to SQL permite consultar directamente los esquemas de base de datos de SQL Server y LINQ to Entities permite consultar un modelo Entity Data Model.

LINQ to DataSet

DataSet es uno de los componentes más usados de ADO.NET, además de un elemento clave del modelo de programación desconectado en el que se basa ADO.NET. En cambio, a pesar de su importancia, DataSet tiene funciones de consulta limitadas.

LINQ to DataSet permite compilar capacidades de consulta más complejas en DataSet mediante la misma funcionalidad de consulta disponible para muchos otros orígenes de datos.

Para más información, vea LINQ to DataSet.

LINQ to SQL

LINQ to SQL proporciona una infraestructura en tiempo de ejecución para administrar los datos relacionales como objetos. En LINQ to SQL, el modelo de datos de una base de datos relacional se asigna a un modelo de objetos expresado en el lenguaje de programación del desarrollador. Cuando se ejecuta la aplicación, LINQ to SQL convierte a SQL las consultas integradas en el lenguaje en el modelo de objetos y las envía a la base de datos para su ejecución. Cuando la base de datos devuelve los resultados, LINQ to SQL los vuelve a convertir en objetos que se pueden manipular.

LINQ to SQL incluye compatibilidad con los procedimientos almacenados y las funciones definidas por el usuario de la base de datos, además de con la herencia en el modelo de objetos.

Para más información, vea LINQ to SQL.

LINQ to Entities

A través del modelo Entity Data Model, los datos relacionales se exponen como objetos en el entorno .NET. Esto hace de la capa de objetos un objetivo idóneo para la compatibilidad con LINQ, ya que permite a los programadores formular consultas en la base de datos con el lenguaje usado para compilar la lógica empresarial. Esta funcionalidad se conoce como LINQ to Entities. Para más información, vea LINQ to Entities.

Vea también

- LINQ y ADO.NET
- Language Integrated Query (LINQ) (C#)

Habilitar un origen de datos para realizar consultas LINQ

18/03/2020 • 7 minutes to read • Edit Online

Hay varias maneras de extender LINQ para permitir consultar cualquier origen de datos en el modelo LINQ El origen de datos podría ser una estructura de datos, un servicio Web, un sistema de archivos o una base de datos, por nombrar algunos. El modelo LINQ facilita a los clientes las consultas a un origen de datos para el que las consultas LINQ están habilitadas, ya que la sintaxis y el modelo de consulta no cambian. Las maneras en las que LINQ se puede extender a estos orígenes de datos son las siguientes:

- Implementar la interfaz IEnumerable<T> en un tipo para habilitar las consultas de LINQ to Objects para ese tipo.
- Crear métodos de operador de consulta estándar como Where y Select que extienden un tipo para habilitar las consultas personalizadas LINQ para ese tipo.
- Crear un proveedor para el origen de datos que implementa la interfaz IQueryable<T>. Un proveedor que implementa esta interfaz recibe consultas LINQ en forma de árboles de expresión, que puede ejecutar de una manera personalizada, por ejemplo, remotamente.
- Crear un proveedor para el origen de datos que aproveche una tecnología de LINQ existente. Este tipo de proveedor permitiría no sólo las consultas, sino también las operaciones de inserción, actualización y eliminación, así como la asignación para tipos definidos por el usuario.

En este tema se analizan estas opciones.

Cómo habilitar las consultas LINQ de un origen de datos

Datos en memoria

Hay dos maneras de habilitar las consultas LINQ para datos en memoria. Si los datos son de un tipo que implementa IEnumerable<T>, puede consultar los datos utilizando LINQ to Objects. Si no tiene sentido habilitar la enumeración de su tipo implementando la interfaz IEnumerable<T>, puede definir métodos de operador de consulta estándar de LINQ en ese tipo o crear métodos de operador de consulta estándar de LINQ que extiendan el tipo. Las implementaciones personalizadas de los operadores de consulta estándar deberían utilizar ejecución diferida para devolver los resultados.

Datos remotos

La mejor opción para permitir las consultas LINQ para un origen de datos remoto consiste en implementar la interfaz IQueryable<T>. Sin embargo, esto es diferente de extender un proveedor como LINQ to SQL para un origen de datos. En Visual Studio 2008, no existen modelos de proveedor para extender tecnologías de LINQ to SQL existentes, como LINQ, a otros tipos de origen de datos.

Proveedores LINQ de IQueryable

Los proveedores de LINQ que implementan IQueryable<T> pueden variar ampliamente en su complejidad. En esta sección se analizan los diferentes niveles de complejidad.

Un proveedor de `IQueryable` menos complejo podría interactuar con un solo método de un servicio Web. Este tipo de proveedor es muy específico, ya que espera información específica en las consultas que administra. Posee un sistema de tipos cerrado, y expone quizá un único tipo de resultado. La mayor parte de la ejecución de la consulta

ocurre localmente, por ejemplo, utilizando las implementaciones Enumerable de los operadores de consulta estándar. Un proveedor menos complejo podría examinar sólo una expresión de llamada a método en el árbol de expresión que representa la consulta, y dejaría que la lógica restante de la consulta se procesara en otra parte.

Un proveedor `IQueryable` de complejidad media podría destinarse a un origen de datos que tuviera un lenguaje de consulta parcialmente expresivo. Si se destina a un servicio Web, podría comunicarse con más de un método del servicio Web y seleccionar el método que va a llamar en función de la pregunta que plantea la consulta. Un proveedor de complejidad media tendría un sistema de tipos más amplio que un proveedor simple, pero todavía sería un sistema de tipos fijo. Por ejemplo, el proveedor podría exponer tipos que tienen relaciones uno a varios y que se pueden recorrer, pero no proporcionaría tecnología de asignación para tipos definidos por el usuario.

Un proveedor `IQueryable` complejo, como el proveedor de LINQ to SQL, podría traducir consultas completas LINQ en un lenguaje de consulta más expresivo, como SQL. Un proveedor complejo es más general que un proveedor menos complejo, porque puede controlar una gran variedad de preguntas en la consulta. También posee un sistema de tipos abierto y, por tanto, debe contener una amplia infraestructura para utilizar tipos definidos por el usuario. El desarrollo de un proveedor complejo requiere un esfuerzo significativo.

Vea también

- IQueryable<T>
- IEnumerable<T>
- Enumerable
- Standard Query Operators Overview (C#)(Información general sobre operadores de consulta estándar (C#))
- LINQ to Objects (C#)

Compatibilidad del IDE y las herramientas de Visual Studio con LINQ (C#)

18/03/2020 • 2 minutes to read • Edit Online

El entorno de desarrollo integrado (IDE) de Visual Studio proporciona las siguientes características que admiten el desarrollo de aplicaciones de LINQ:

Object Relational Designer

Object Relational Designer es una herramienta de diseño visual que puede usar en aplicaciones de LINQ to SQL para generar clases en C# que representan los datos relacionales en una base de datos subyacente. Para obtener más información, vea LINQ to SQL Tools en Visual Studio.

Herramienta de línea de comandos SQLMetal

SQLMetal es una herramienta de línea de comandos que puede usarse en procesos de compilación para generar clases de bases de datos existentes para su uso en aplicaciones de LINQ to SQL. Para obtener más información, vea SqlMetal.exe (Herramienta de generación de código).

Editor de código compatible con LINQ

El editor de código de C# admite LINQ de manera extensiva con IntelliSense y las funciones de formato.

Compatibilidad con el depurador de Visual Studio

El depurador de Visual Studio admite la depuración de expresiones de consulta. Para obtener más información, vea Depurar LINQ.

Vea también

- Language Integrated Query (LINQ) (C#)

Programación orientada a objetos (C#)

29/06/2020 • 19 minutes to read • Edit Online

C# proporciona compatibilidad completa para la programación orientada a objetos incluida la encapsulación, la herencia y el polimorfismo.

- La *encapsulación* significa que un grupo de propiedades, métodos y otros miembros relacionados se tratan como una sola unidad u objeto.
- La *herencia* describe la posibilidad de crear nuevas clases basadas en una clase existente.
- El *polimorfismo* significa que puede tener múltiples clases que se pueden usar de manera intercambiable, aunque cada clase implementa las mismas propiedades o los mismos métodos de maneras diferentes.

Clases y objetos

Los términos *clase* y *objeto* describen el *tipo* de los objetos y las *instancias* de las clases, respectivamente. Así, la acción de crear un objeto se denomina *creación de instancias*. Con la analogía de plano, una clase es un plano y un objeto es un edificio construido a partir de ese plano.

Para definir una clase:

```
class SampleClass
{
}
```

En C# también se proporcionan tipos denominados *estructuras* que son útiles cuando no se necesita compatibilidad con la herencia o el polimorfismo.

Para definir una estructura:

```
struct SampleStruct
{
}
```

Para obtener más información, vea los artículos sobre las palabras clave class y struct.

Miembros de la clase

Cada clase puede tener distintos *miembros de clase*, entre los que se incluyen las propiedades que describen los datos de clase, los métodos que definen el comportamiento de la clase y los eventos que proporcionan comunicación entre distintos objetos y clases.

Propiedades y campos

Los campos y propiedades representan información que contiene un objeto. Los campos se parecen a las variables ya que se pueden leer o establecer directamente, sujetos a los modificadores de acceso aplicables.

Para definir un campo al que se pueda acceder desde dentro de las instancias de la clase:

```
public class SampleClass
{
    string sampleField;
}
```

Las propiedades tienen descriptores de acceso `get` y `set`, que proporcionan más control sobre la forma en que se establecen o devuelven los valores.

C# permite crear un campo privado para almacenar el valor de propiedad o usar propiedades de implementación automática que crean este campo en segundo plano automáticamente y proporcionan la lógica básica para los procedimientos de propiedad.

Para definir una propiedad implementada automáticamente:

```
class SampleClass
{
    public int SampleProperty { get; set; }
}
```

Si necesita realizar algunas operaciones adicionales para leer y escribir el valor de propiedad, defina un campo para almacenar el valor de propiedad y proporcione la lógica básica para almacenarlo y recuperar lo:

```
class SampleClass
{
    private int _sample;
    public int Sample
    {
        // Return the value stored in a field.
        get => _sample;
        // Store the value in the field.
        set =>  _sample = value;
    }
}
```

La mayoría de las propiedades tienen métodos o procedimientos tanto para establecer como para obtener el valor de propiedad. Sin embargo, se pueden crear propiedades de solo lectura o solo escritura para restringir su modificación o lectura. En C#, se puede omitir el método de propiedad `get` o `set`. Pero las propiedades implementadas automáticamente no pueden ser de solo escritura. Las propiedades implementadas automáticamente de solo lectura se pueden establecer en constructores de la clase contenedora.

Para obtener más información, consulte:

- get
- set

Métodos

Un *método* es una acción que un objeto puede realizar.

Para definir un método de una clase:

```
class SampleClass
{
    public int SampleMethod(string sampleParam)
    {
        // Insert code here
    }
}
```

Una clase puede tener varias implementaciones o *sobrecargas* del mismo método que se diferencian en el número de parámetros o de tipos de parámetro.

Para sobrecargar un método:

```
public int SampleMethod(string sampleParam) { }
public int SampleMethod(int sampleParam) { }
```

En la mayoría de los casos, un método se declara dentro de una definición de clase. En cambio, C# también admite los *métodos de extensión*, que le permiten agregar métodos a una clase existente fuera de la definición de la clase actual.

Para obtener más información, consulte:

- Métodos
- Métodos de extensión

Constructores

Los constructores son métodos de clase que se ejecutan automáticamente cuando se crea un objeto de un tipo determinado. Normalmente, los constructores inicializan los miembros de datos del nuevo objeto. Un constructor solo puede ejecutarse una vez cuando se crea una clase. Además, el código del constructor siempre se ejecuta antes que cualquier otro código en una clase. Sin embargo, puede crear varias sobrecargas del constructor de la misma forma que para cualquier otro método.

Para definir un constructor para una clase:

```
public class SampleClass
{
    public SampleClass()
    {
        // Add code here
    }
}
```

Para obtener más información, vea Constructores.

Finalizadores

Los finalizadores se usan para destruir instancias de clases. En .NET, el recolector de elementos no utilizados administra automáticamente la asignación y la liberación de memoria para los objetos administrados en la aplicación. En cambio, es posible que aún se necesiten finalizadores para limpiar cualquiera de los recursos no administrados creados por la aplicación. Solo puede haber un finalizador para una clase.

Para obtener más información sobre los finalizadores y la recolección de elementos no utilizados en .NET, vea Recolección de elementos no utilizados.

Events

Cuando ocurre algo interesante, los eventos habilitan una clase u objeto para notificarlo a otras clases u objetos. La clase que envía (o genera) el evento recibe el nombre de *publicador* y las clases que reciben (o controlan) el evento se denominan *suscriptores*. Para obtener más información sobre los eventos y la forma en que se generan y controlan, vea Eventos.

- Para declarar un evento en una clase, use la palabra clave event.
- Para generar un evento, invoque al delegado de eventos.
- Para suscribirse a un evento, use el operador `+=`; para anular la suscripción de un evento, use el operador `-=`.

Clases anidadas

Una clase definida dentro de otra se denomina *anidada*. De forma predeterminada, una clase anidada es privada.

```
class Container
{
    class Nested
    {
        // Add code here.
    }
}
```

Para crear una instancia de la clase anidada, use el nombre de la clase contenedora seguido de un punto y seguido, a continuación, del nombre de la clase anidada:

```
Container.Nested nestedInstance = new Container.Nested()
```

Modificadores y niveles de acceso

Todas las clases y miembros de clase pueden especificar el nivel de acceso que proporcionan a otras clases mediante los *modificadores de acceso*.

Están disponibles los siguientes modificadores de acceso:

MODIFICADOR DE C#	DEFINICIÓN
public	Puede obtener acceso al tipo o miembro cualquier otro código del mismo ensamblado o de otro ensamblado que haga referencia a éste.
private	Solamente puede obtener acceso al tipo o miembro el código de la misma clase.
protected	Solamente puede obtener acceso al tipo o miembro el código de la misma clase o de una clase derivada.
internal	Puede obtener acceso al tipo o miembro cualquier código del mismo ensamblado, pero no de un ensamblado distinto.
protected internal	Puede obtener acceso al tipo o miembro cualquier código del mismo ensamblado o cualquier clase derivada de otro ensamblado.
private protected	Un código de la misma clase o de una clase derivada dentro del ensamblado de clase base puede acceder al tipo o miembro en cuestión.

Para obtener más información, consulte Modificadores de acceso.

Creación de instancias de clases

Para crear un objeto, debe crear una o varias instancias de una clase.

```
SampleClass sampleObject = new SampleClass();
```

Una vez creadas las instancias de una clase, puede asignar valores a las propiedades y los campos de la instancia, así como invocar métodos de clase.

```
// Set a property value.
sampleObject.sampleProperty = "Sample String";
// Call a method.
sampleObject.SampleMethod();
```

Para asignar valores a las propiedades durante el proceso de creación de instancias de una clase, use los inicializadores de objeto:

```
// Set a property value.
var sampleObject = new SampleClass
{
    FirstProperty = "A",
    SecondProperty = "B"
};
```

Para obtener más información, consulte:

- new (operador)
- Inicializadores de objeto y colección

Clases y miembros estáticos

Un miembro estático de la clase es una propiedad, un procedimiento o un campo que comparten todas las instancias de una clase.

Para definir un miembro estático:

```
static class SampleClass
{
    public static string SampleString = "Sample String";
}
```

Para obtener acceso al miembro estático, use el nombre de la clase sin crear un objeto perteneciente a esta:

```
Console.WriteLine(SampleClass.SampleString);
```

Las clases estáticas en C# solo tienen miembros estáticos y no se puede crear una instancia de ellas. Además, los miembros estáticos no pueden tener acceso a las propiedades, los campos o los métodos no estáticos.

Para obtener más información, vea: static.

Tipos anónimos

Los tipos anónimos permiten crear objetos sin escribir una definición de clase para el tipo de datos. En su lugar, el compilador genera una clase. La clase no tiene ningún nombre que se pueda usar y contiene las propiedades especificadas al declarar el objeto.

Para crear una instancia de un tipo anónimo:

```
// sampleObject is an instance of a simple anonymous type.
var sampleObject = new
{
    FirstProperty = "A",
    SecondProperty = "B"
};
```

Para obtener más información, consulte: Tipos anónimos.

Herencia

La herencia permite crear una nueva clase que reutiliza, extiende y modifica el comportamiento que se define en otra clase. La clase cuyos miembros se heredan se denomina *clase base* y la clase que hereda esos miembros se denomina *clase derivada*. En cambio, todas las clases de C# heredan implícitamente de la clase Object que admite la jerarquía de clases .NET y proporciona servicios de bajo nivel a todas las clases.

> **NOTE**
> C# no admite la herencia múltiple. Es decir, solo puede especificar una clase base para una clase derivada.

Para heredar de una clase base:

```
class DerivedClass:BaseClass { }
```

De forma predeterminada, todas las clases se pueden heredar. Sin embargo, puede especificar si una clase no se debe usar como clase base o bien crear una clase que solo se pueda usar como clase base.

Para especificar que una clase no se puede usar como clase base:

```
public sealed class A { }
```

Para especificar que una clase se puede usar solo como clase base y no se pueden crear instancias de esta:

```
public abstract class B { }
```

Para obtener más información, consulte:

- sealed
- abstract

Reemplazar miembros

De forma predeterminada, una clase derivada hereda todos los miembros de su clase base. Si desea cambiar el comportamiento del miembro heredado, debe invalidarlo. Es decir, se puede definir una nueva implementación del método, la propiedad o el evento en la clase derivada.

Los siguientes modificadores se utilizan para controlar cómo se reemplazan propiedades y métodos:

MODIFICADOR DE C#	DEFINICIÓN
virtual	Permite invalidar un miembro de una clase derivada.
override	Invalida un miembro virtual (invalidable) definido en la clase base.
abstract	Requiere que se invalide un miembro de clase en la clase derivada.
new (modificador)	Oculta un miembro heredado de una clase base.

Interfaces

Las interfaces, como las clases, definen un conjunto de propiedades, métodos y eventos. Pero de forma contraria a

las clases, las interfaces no proporcionan implementación. Se implementan como clases y se definen como entidades separadas de las clases. Una interfaz representa un contrato, en el cual una clase que implementa una interfaz debe implementar cualquier aspecto de dicha interfaz exactamente como esté definido.

Para definir una interfaz:

```
interface ISampleInterface
{
    void DoSomething();
}
```

Para implementar una interfaz en una clase:

```
class SampleClass : ISampleInterface
{
    void ISampleInterface.DoSomething()
    {
        // Method implementation.
    }
}
```

Para obtener más información, vea el artículo de la guía de programación sobre interfaces y el artículo de la referencia del lenguaje sobre la palabra clave interface.

Genéricos

Las clases, las estructuras, las interfaces y los métodos de .NET pueden incluir *parámetros de tipo* que definen los tipos de objetos que estos pueden almacenar o usar. El ejemplo más común de elementos genéricos es una colección, donde se puede especificar el tipo de objetos que se va a almacenar en una colección.

Para definir una clase genérica:

```
public class SampleGeneric<T>
{
    public T Field;
}
```

Para crear una instancia de una clase genérica:

```
var sampleObject = new SampleGeneric<string>();
sampleObject.Field = "Sample string";
```

Para obtener más información, consulte:

- Elementos genéricos en .NET
- Aspectos genéricos: Guía de programación de C#

Delegados

Un *delegado* es un tipo que define una firma de método y que puede proporcionar una referencia a cualquier método con una firma compatible. Puede invocar (o llamar) al método a través del delegado. Los delegados se utilizan para pasar métodos como argumentos a otros métodos.

> **NOTE**
> Los controladores de eventos no son más que métodos que se invocan a través de delegados. Para obtener más información sobre el uso de delegados en el control de eventos, vea Eventos.

Para crear un delegado:

```
public delegate void SampleDelegate(string str);
```

Para crear una referencia a un método que coincida con la firma especificada por el delegado:

```
class SampleClass
{
    // Method that matches the SampleDelegate signature.
    public static void SampleMethod(string message)
    {
        // Add code here.
    }

    // Method that instantiates the delegate.
    void SampleDelegate()
    {
        SampleDelegate sd = sampleMethod;
        sd("Sample string");
    }
}
```

Para obtener más información, vea el artículo de la guía de programación sobre Delegados y el artículo de la referencia del lenguaje sobre la palabra clave delegate.

Vea también

- Guía de programación de C#

Reflexión (C#)

18/03/2020 • 3 minutes to read • Edit Online

La reflexión proporciona objetos (de tipo Type) que describen los ensamblados, módulos y tipos. Puede usar la reflexión para crear dinámicamente una instancia de un tipo, enlazar el tipo a un objeto existente u obtener el tipo desde un objeto existente e invocar sus métodos, o acceder a sus campos y propiedades. Si usa atributos en el código, la reflexión le permite acceder a ellos. Para obtener más información, consulte Attributes (Atributos).

Este es un ejemplo simple de reflexión que usa el método GetType(), heredado por todos los tipos de la clase base `Object`, para obtener el tipo de una variable:

> **NOTE**
>
> Asegúrese de agregar `using System;` y `using System.Reflection;` en la parte superior del archivo de *.cs*.

```
// Using GetType to obtain type information:
int i = 42;
Type type = i.GetType();
Console.WriteLine(type);
```

El resultado es `System.Int32`.

En el ejemplo siguiente se usa la reflexión para obtener el nombre completo del ensamblado cargado.

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

El resultado es `System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`.

> **NOTE**
>
> Las palabras clave de C# `protected` y `internal` no tienen ningún significado en IL y no se usan en las API de reflexión. Los términos correspondientes en IL son *Family* y *Assembly*. Para identificar un método `internal` con la reflexión, use la propiedad IsAssembly. Para identificar un método `protected internal`, use IsFamilyOrAssembly.

Información general de la reflexión

La reflexión resulta útil en las siguientes situaciones:

- Cuando tenga que acceder a atributos en los metadatos del programa. Para obtener más información, consulte Recuperar información almacenada en atributos.
- Para examinar y crear instancias de tipos en un ensamblado.
- Para generar nuevos tipos en tiempo de ejecución. Usar clases en System.Reflection.Emit.
- Para llevar a cabo métodos de acceso de enlace en tiempo de ejecución en tipos creados en tiempo de ejecución. Consulte el tema Dynamically Loading and Using Types (Cargar y usar tipos dinámicamente).

Secciones relacionadas

Para obtener más información:

- Reflexión
- Ver información tipos
- Reflection and Generic Types (Reflexión y tipos genéricos)
- System.Reflection.Emit
- Retrieving Information Stored in Attributes (Recuperar la información almacenada en atributos)

Vea también

- Guía de programación de C#
- Ensamblados de .NET

Serialización (C#)

29/06/2020 • 9 minutes to read • Edit Online

La serialización es el proceso de convertir un objeto en una secuencia de bytes para almacenarlo o transmitirlo a la memoria, a una base de datos o a un archivo. Su propósito principal es guardar el estado de un objeto para poder volver a crearlo cuando sea necesario. El proceso inverso se denomina deserialización.

Funcionamiento de la serialización

En esta ilustración se muestra el proceso general de la serialización:

El objeto se serializa en una secuencia que incluye los datos. La secuencia también puede tener información sobre el tipo del objeto, como la versión, la referencia cultural y el nombre del ensamblado. A partir de esa secuencia, el objeto se puede almacenar en una base de datos, en un archivo o en memoria.

Usos de la serialización

La serialización permite al desarrollador guardar el estado de un objeto y volver a crearlo según sea necesario, ya que proporciona almacenamiento de los objetos e intercambio de datos. A través de la serialización, un desarrollador puede realizar acciones como las siguientes:

- Enviar el objeto a una aplicación remota mediante un servicio web
- Pasar un objeto de un dominio a otro
- Pasar un objeto a través de un firewall como una cadena JSON o XML
- Mantener la seguridad o información específica del usuario entre aplicaciones

Serialización de JSON

El espacio de nombres System.Text.Json contiene clases para la serialización y deserialización de notación de objetos JavaScript (JSON). JSON es un estándar abierto que se usa normalmente para compartir datos en la web.

La serialización de JSON serializa las propiedades públicas de un objeto en una cadena, una matriz de bytes o una secuencia que se ajusta a la especificación de JSON RFC 8259. Para controlar la forma en que JsonSerializer serializa o deserializa una instancia de la clase:

- Use un objeto JsonSerializerOptions.
- Aplique atributos del espacio de nombres System.Text.Json.Serialization a clases o propiedades.
- Implemente convertidores personalizados.

Serialización XML y binaria

El espacio de nombres System.Runtime.Serialization contiene clases para la serialización y deserialización binaria y XML.

La serialización binaria utiliza la codificación binaria para generar una serialización compacta para usos como almacenamiento o secuencias de red basadas en socket. En la serialización binaria se serializan todos los miembros, incluso aquellos que son de solo lectura, y mejora el rendimiento.

La serialización XML serializa las propiedades y los campos públicos de un objeto o los parámetros y valores devueltos de los métodos en una secuencia XML que se ajusta a un documento específico del lenguaje de definición de esquema XML (XSD). La serialización XML produce clases fuertemente tipadas cuyas propiedades y campos públicos se convierten a XML. System.Xml.Serialization contiene clases para serializar y deserializar XML. Se aplican atributos a clases y a miembros de clase para controlar la forma en que XmlSerializer serializa o deserializa una instancia de la clase.

Conversión de un objeto en serializable

Para la serialización binaria o XML, necesita lo siguiente:

- Objeto que se va a serializar
- Secuencia para incluir el objeto serializado
- Instancia de System.Runtime.Serialization.Formatter

Aplique el atributo SerializableAttribute a un tipo para indicar que se pueden serializar instancias de este tipo. Si se intenta serializar pero el tipo no tiene el atributo SerializableAttribute, se produce una excepción.

Para evitar que un campo se serialice, aplique el atributo NonSerializedAttribute. Si un campo de un tipo serializable contiene un puntero, un controlador o alguna otra estructura de datos específica para un entorno concreto y el campo no se puede reconstituir correctamente en un entorno diferente, puede convertirlo en no serializable.

Si una clase serializada contiene referencias a objetos de otras clases marcadas como SerializableAttribute, esos objetos también se serializarán.

Serialización básica y personalizada

La serialización binaria y XML se puede realizar de dos formas: de manera básica y personalizada.

La serialización básica utiliza .NET para serializar automáticamente el objeto. El único requisito es que la clase tenga el atributo SerializableAttribute aplicado. NonSerializedAttribute puede usarse para impedir la serialización de campos específicos.

Cuando se usa la serialización básica, el control de versiones de objetos puede causar problemas. Use la serialización personalizada si los problemas de control de versiones son importantes. La serialización básica es la manera más fácil de realizar la serialización, pero no proporciona mucho control sobre el proceso.

En la serialización personalizada, puede especificar exactamente qué objetos se serializarán y cómo se llevará a cabo la serialización. La clase debe marcarse como SerializableAttribute e implementar la interfaz ISerializable. Si quiere que el objeto también se deserialice de forma personalizada, use un constructor personalizado.

Serialización de diseñador

La serialización de diseñador es una forma especial de serialización que conlleva el tipo de persistencia de objeto asociado a las herramientas de desarrollo. La serialización de diseñador es un proceso que consiste en convertir un gráfico de objetos en un archivo de código fuente que puede utilizarse posteriormente para recuperar el gráfico de objetos. Un archivo de código fuente puede contener código, marcado o incluso información de la tabla SQL.

Temas relacionados y ejemplos

En Información general de System.Text.Json se muestra cómo obtener la biblioteca `System.Text.Json`.

En Procedimiento para serializar y deserializar JSON en .NET se muestra cómo leer y escribir datos de objetos a y

desde JSON mediante la clase JsonSerializer.

Tutorial: Conservar un objeto en Visual Studio (C#)
Se explica cómo se puede usar la serialización para conservar los datos de un objeto entre instancias, lo que le permite almacenar valores y recuperarlos la próxima vez que se cree una instancia del objeto.

Procedimiento para leer datos de objeto de un archivo XML (C#)
Se muestra cómo leer los datos de objetos que se han escrito anteriormente en un archivo XML con la clase XmlSerializer.

Procedimiento para escribir datos de objeto en un archivo XML (C#)
Se muestra cómo escribir el objeto de una clase en un archivo XML con la clase XmlSerializer.

Procedimiento para escribir datos de objeto en un archivo XML (C#)

29/06/2020 • 2 minutes to read • Edit Online

En este ejemplo se escribe el objeto de una clase en un archivo XML con la clase XmlSerializer.

Ejemplo

```
public class XMLWrite
{

   static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
"//SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

Compilar el código

La clase que se serializa debe tener un constructor público sin parámetros.

Programación sólida

Las condiciones siguientes pueden provocar una excepción:

- La clase que se está serializando no tiene un constructor público sin parámetros.
- El archivo ya existe y es de solo lectura (IOException).
- La ruta de acceso del archivo es demasiado larga (PathTooLongException).
- El disco está lleno (IOException).

Seguridad de .NET

En este ejemplo se crea un nuevo archivo, si este no existe aún. Si una aplicación necesita crear un archivo, precisará acceso `Create` para la carpeta. Si el archivo ya existe, la aplicación necesitará solo acceso `Write`, un privilegio menor. Siempre que sea posible, resulta más seguro crear el archivo durante la implementación y conceder solo acceso `Read` a un único archivo, en lugar de acceso `Create` para una carpeta.

Vea también

- StreamWriter
- Procedimiento para leer datos de objeto de un archivo XML (C#)
- Serialización (C#)

Procedimiento para leer datos de objeto de un archivo XML (C#)

29/06/2020 • 2 minutes to read • Edit Online

En este ejemplo se leen los datos de objetos que se han escrito anteriormente en un archivo XML con la clase XmlSerializer.

Ejemplo

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview =  (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);

}
```

Compilar el código

Reemplace el nombre de archivo "c:\temp\SerializationOverview.xml" por el nombre del archivo que contiene los datos serializados. Para más información sobre la serialización de datos, consulte Procedimiento para escribir datos de objeto en un archivo XML (C#).

La clase debe tener un constructor público sin parámetros.

Solo se deserializan las propiedades y los campos públicos.

Programación sólida

Las condiciones siguientes pueden provocar una excepción:

- La clase que se está serializando no tiene un constructor público sin parámetros.
- Los datos del archivo no representan los datos de la clase que se va a deserializar.
- El archivo no existe (IOException).

Seguridad de .NET

Compruebe siempre las entradas y nunca deserialice datos de un origen que no sea de confianza. El objeto que se ha vuelto a crear se ejecuta en un equipo local con los permisos del código que lo ha deserializado. Compruebe todas las entradas antes de utilizar los datos en la aplicación.

Vea también

- StreamWriter
- Procedimiento para escribir datos de objeto en un archivo XML (C#)
- Serialización (C#)
- Guía de programación de C#

Tutorial: Conservar un objeto con C#

29/06/2020 • 9 minutes to read • Edit Online

Puede usar la serialización para conservar los datos de un objeto entre instancias, lo que le permite almacenar valores y recuperarlos la próxima vez que se cree una instancia del objeto.

En este tutorial, creará un objeto `Loan` básico y conservará sus datos en un archivo. Después, recuperará los datos del archivo cuando vuelva a crear el objeto.

> **IMPORTANT**
>
> En este ejemplo se crea un nuevo archivo, si este no existe aún. Si una aplicación debe crear un archivo, es necesario que tenga el permiso `Create` en la carpeta. Los permisos se establecen mediante el uso de las listas de control de acceso. Si el archivo ya existe, la aplicación necesitará solo un permiso `Write`, un permiso menor. Siempre que sea posible, resulta más seguro crear el archivo durante la implementación y conceder solo permisos `Read` a un único archivo (en lugar de crear permisos para una carpeta). También es más seguro escribir datos en carpetas de usuario en lugar de hacerlo en la carpeta raíz o en la carpeta Archivos de programa.

> **IMPORTANT**
>
> En este ejemplo se almacenan datos en un archivo de formato binario. Estos formatos no deben usarse para datos confidenciales, como contraseñas o información de tarjetas de crédito.

Requisitos previos

- Para compilar y ejecutar, instalar el SDK de .NET Core.
- Instale el editor de código que prefiera si aún no lo ha hecho.

> **TIP**
>
> ¿Es necesario instalar un editor de código? Pruebe Visual Studio.

- El ejemplo requiere C# 7.3. Consulte Seleccionar la versión del lenguaje C#.

En el repositorio de GitHub de ejemplos de .NET puede examinar el código de ejemplo en línea.

Crear el objeto Loan

El primer paso consiste en crear una clase `Loan` y una aplicación de consola que use la clase:

1. Cree una aplicación. Escriba `dotnet new console -o serialization` para crear una aplicación de consola en un subdirectorio llamado `serialization`.
2. Abra la aplicación en el editor y agregue una nueva clase denominada `Loan.cs`.
3. Agregue el siguiente código a la clase `Loan`:

```
public class Loan : INotifyPropertyChanged
 {
     public double LoanAmount { get; set; }
     public double InterestRatePercent { get; set; }

     [field:NonSerialized()]
     public DateTime TimeLastLoaded { get; set; }

     public int Term { get; set; }

     private string customer;
     public string Customer
     {
         get { return customer; }
         set
         {
             customer = value;
             PropertyChanged?.Invoke(this,
               new PropertyChangedEventArgs(nameof(Customer)));
         }
     }

     [field: NonSerialized()]
     public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

     public Loan(double loanAmount,
                 double interestRate,
                 int term,
                 string customer)
     {
         this.LoanAmount = loanAmount;
         this.InterestRatePercent = interestRate;
         this.Term = term;
         this.customer = customer;
     }
 }
```

También tendrá que crear una aplicación que use la clase `Loan`.

Serializar el objeto Loan

1. Abra `Program.cs`. Agregue el código siguiente:

```
Loan TestLoan = new Loan(10000.0, 7.5, 36, "Neil Black");
```

Agregue un controlador de eventos del evento `PropertyChanged` y unas cuantas líneas para modificar el objeto `Loan` y ver los cambios. En el siguiente código puede ver las adiciones realizadas:

```
TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRatePercent);
TestLoan.InterestRatePercent = 7.1;
Console.WriteLine(TestLoan.InterestRatePercent);
```

Llegado este punto, puede ejecutar el código y ver el resultado actual:

```
New customer value: Henry Clay
7.5
7.1
```

Si ejecuta esta aplicación repetidamente, verá que siempre escribe los mismos valores. Se creará un objeto Loan cada vez que ejecute el programa. En el mundo real, las tasas de interés cambian periódicamente, pero no necesariamente cada vez que se ejecuta la aplicación. El código de serialización conlleva que se va a conservar la tasa de interés más reciente entre las instancias de la aplicación. En el paso siguiente, hará esto agregando la serialización a la clase Loan.

Usar la serialización para conservar el objeto

Para conservar los valores de la clase Loan, primero debe marcar la clase con el atributo `Serializable`. Agregue el siguiente código encima de la declaración de la clase Loan:

```
[Serializable()]
```

SerializableAttribute indica al compilador que todo el contenido de la clase se puede conservar en un archivo. El evento `PropertyChanged` no representa la parte del gráfico de objeto que se debe almacenar, de modo que no se debería serializar. Si lo hace, se serializarían todos los objetos que estén asociados a ese evento. Puede agregar NonSerializedAttribute a la declaración de campo del controlador de eventos `PropertyChanged`.

```
[field: NonSerialized()]
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

A partir de C# 7.3, los atributos se pueden asociar al campo de respaldo de una propiedad implementada automáticamente usando el valor de destino `field`. El siguiente código agrega una propiedad `TimeLastLoaded` y la marca como no serializable:

```
[field:NonSerialized()]
public DateTime TimeLastLoaded { get; set; }
```

El siguiente paso consiste en agregar el código de serialización a la aplicación LoanApp. Para serializar la clase y escribirla en un archivo, usaremos los espacios de nombres System.IO y System.Runtime.Serialization.Formatters.Binary. Para evitar escribir los nombres completos, puede agregar referencias a los espacios de nombres necesarios, tal y como se muestra en el siguiente código:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

El siguiente paso es agregar código para deserializar el objeto del archivo cuando el objeto se crea. Agregue una constante a la clase para el nombre de archivo de los datos serializados, tal y como se muestra en el siguiente código:

```
const string FileName = @"../../../SavedLoan.bin";
```

Luego, agregue el siguiente código después de la línea que crea el objeto `TestLoan`:

```
if (File.Exists(FileName))
{
    Console.WriteLine("Reading saved file");
    Stream openFileStream = File.OpenRead(FileName);
    BinaryFormatter deserializer = new BinaryFormatter();
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);
    TestLoan.TimeLastLoaded = DateTime.Now;
    openFileStream.Close();
}
```

Primero hay que confirmar que el archivo existe. Si existe, cree una clase Stream para leer el archivo binario y una clase BinaryFormatter para traducirlo. También necesita convertir del tipo de secuencia al tipo de objeto Loan.

Luego, debemos agregar código para serializar la clase en un archivo. Agregue el siguiente código después del código existente en el método `Main`:

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

En este punto, podrá compilar y ejecutar la aplicación de nuevo. La primera vez que se ejecuta, observe que el tipo de interés comienza en 7.5 y, después, pasa a 7.1. Cierre la aplicación y, después, ejecútela de nuevo. Ahora, la aplicación muestra un mensaje que indica que se ha leído el archivo guardado y la tasa de interés es 7.1 incluso antes del código que la cambia.

Vea también

- Serialización (C#)
- Guía de programación de C#

Instrucciones, expresiones y operadores (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

El código de C# que conforma una aplicación consta de instrucciones basadas en palabras clave, expresiones y operadores. Esta sección contiene información sobre los elementos fundamentales de un programa de C#.

Para obtener más información, consulte:

- Instrucciones
- Expresiones
  - Miembros con forma de expresión
- Funciones anónimas
- Comparaciones de igualdad

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Conversiones de tipos

Instrucciones (Guía de programación de C#)

18/03/2020 • 11 minutes to read • Edit Online

Las acciones que realiza un programa se expresan en instrucciones. Entre las acciones comunes se incluyen declarar variables, asignar valores, llamar a métodos, recorrer colecciones en bucle y crear una bifurcación a uno u otro bloque de código, en función de una condición determinada. El orden en el que se ejecutan las instrucciones en un programa se denomina flujo de control o flujo de ejecución. El flujo de control puede variar cada vez que se ejecuta un programa, en función de cómo reacciona el programa a la entrada que recibe en tiempo de ejecución.

Una instrucción puede constar de una sola línea de código que finaliza en un punto y coma o de una serie de instrucciones de una sola línea en un bloque. Un bloque de instrucciones se incluye entre llaves {} y puede contener bloques anidados. En el código siguiente se muestran dos ejemplos de instrucciones de una sola línea y un bloque de instrucciones de varias líneas:

```
    static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment statement:
        int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
                                     counter, radius, circumference);

            // Expression statement (postfix increment).
            counter++;
        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

Tipos de instrucciones

En la tabla siguiente se muestran los distintos tipos de instrucciones de C# y sus palabras clave asociadas, con vínculos a temas que incluyen más información:

CATEGORÍA	PALABRAS CLAVE DE C# / NOTAS
Instrucciones de declaración	Una instrucción de declaración introduce una variable o constante nueva. Una declaración de variable puede asignar opcionalmente un valor a la variable. En una declaración de constante, se requiere la asignación.
Instrucciones de expresión	Las instrucciones de expresión que calculan un valor deben almacenar el valor en una variable. Para obtener más información, vea Instrucciones de expresión.
Instrucciones de selección	Las instrucciones de selección permiten crear bifurcaciones a diferentes secciones de código, en función de una o varias condiciones especificadas. Para obtener más información, consulta los temas siguientes: • if • else • switch • case
Instrucciones de iteración	Las instrucciones de iteración permiten recorrer en bucle colecciones, como matrices, o realizar el mismo conjunto de instrucciones repetidas veces hasta que se cumpla una condición especificada. Para obtener más información, consulta los temas siguientes: • do • for • foreach • in • while
Instrucciones de salto	Las instrucciones de salto transfieren el control a otra sección de código. Para obtener más información, consulta los temas siguientes: • break • continue • default • goto • return • yield
Instrucciones para el control de excepciones	Las instrucciones para el control de excepciones permiten recuperarse correctamente de condiciones excepcionales producidas en tiempo de ejecución. Para obtener más información, consulta los temas siguientes: • throw • try-catch • try-finally • try-catch-finally

CATEGORÍA	PALABRAS CLAVE DE C# / NOTAS
Checked y unchecked	Las instrucciones checked y unchecked permiten especificar si las operaciones numéricas pueden producir un desbordamiento cuando el resultado se almacena en una variable que es demasiado pequeña para contener el valor resultante. Para obtener más información, vea checked y unchecked.
Instrucción `await`	Si marca un método con el modificador async , puede usar el operador await en el método. Cuando el control alcanza una expresión `await` en el método asincrónico, el control se devuelve al autor de llamada y el progreso del método se suspende hasta que se completa la tarea esperada. Cuando se completa la tarea, la ejecución puede reanudarse en el método. Para obtener un ejemplo sencillo, vea la sección "Métodos asincrónicos" de Métodos. Para obtener más información, vea Programación asincrónica con Async y Await.
Instrucción `yield return`	Un iterador realiza una iteración personalizada en una colección, como una lista o matriz. Un iterador usa la instrucción yield return para devolver cada elemento de uno en uno. Cuando se alcanza una instrucción `yield return`, se recuerda la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama el iterador. Para obtener más información, consulta Iteradores.
Instrucción `fixed`	La instrucción fixed impide que el recolector de elementos no utilizados cambie la ubicación de una variable móvil. Para obtener más información, vea fixed.
Instrucción `lock`	La instrucción lock permite limitar el acceso a bloques de código a un solo subproceso de cada vez. Para obtener más información, vea lock.
Instrucciones con etiqueta	Puede asignar una etiqueta a una instrucción y, después, usar la palabra clave goto para saltar a la instrucción con etiqueta. (Vea el ejemplo de la línea siguiente).
Instrucción vacía	La instrucción vacía consta únicamente de un punto y coma. No hace nada y se puede usar en lugares en los que se requiere una instrucción, pero no es necesario realizar ninguna acción.

Instrucciones de declaración

En el código siguiente se muestran ejemplos de declaraciones de variables con y sin una asignación inicial, y una declaración constante con la inicialización necesaria.

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Instrucciones de expresión

En el código siguiente se muestran ejemplos de instrucciones de expresión, que incluyen la asignación, la creación de objetos con asignación y la invocación de método.

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

Instrucción vacía

En los ejemplos siguientes se muestran dos usos de una instrucción vacía:

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

Instrucciones insertadas

Algunas instrucciones, incluidas do, while, for y foreach, siempre van seguidas de una instrucción insertada. Esta instrucción insertada puede ser una sola instrucción o varias instrucciones incluidas entre llaves {} en un bloque de instrucciones. Las instrucciones insertadas de una sola línea también pueden ir entre llaves {}, como se muestra en el siguiente ejemplo:

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

Una instrucción insertada que no está incluida entre llaves {} no puede ser una instrucción de declaración o una instrucción con etiqueta. Esto se muestra en el ejemplo siguiente:

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Coloque la instrucción insertada en un bloque para solucionar el error:

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

Bloques de instrucciones anidadas

Los bloques de instrucciones pueden anidarse, como se muestra en el código siguiente:

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

Instrucciones inaccesibles

Si el compilador determina que el flujo de control no puede alcanzar nunca una instrucción determinada bajo ninguna circunstancia, producirá una advertencia CS0162, como se muestra en el ejemplo siguiente:

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

especificación del lenguaje C#

Para más información, vea la sección Instrucciones de la especificación del lenguaje C#.

Vea también

- Guía de programación de C#
- Palabras clave de instrucciones
- Expresiones

Expresiones (Guía de programación de C#)

18/03/2020 • 9 minutes to read • Edit Online

Una *expresión* es una secuencia de uno o más operandos y cero o más operadores que se pueden evaluar como un valor, objeto, método o espacio de nombres único. Las expresiones pueden constar de un valor literal, una invocación de método, un operador y sus operandos o un *nombre simple*. Los nombres simples pueden ser el nombre de una variable, el miembro de un tipo, el parámetro de un método, un espacio de nombres o un tipo.

Las expresiones pueden usar operadores que, a su vez, usan otras expresiones como parámetros o llamadas a métodos cuyos parámetros son, a su vez, otras llamadas a métodos, de modo que pueden variar de simples a muy complejas. A continuación se muestran dos ejemplos de expresiones:

```
((x < 10) && ( x > 5)) || ((x > 20) && (x < 25));

System.Convert.ToInt32("35");
```

Valores de expresión

En la mayoría de los contextos donde se usan expresiones, por ejemplo en instrucciones o parámetros de método, se espera que la expresión se evalúe como algún valor. Si x e y son enteros, la expresión `x + y` se evalúa como un valor numérico. La expresión `new MyClass()` se evalúa como una referencia a una nueva instancia de una clase `MyClass`. La expresión `myClass.ToString()` se evalúa como una cadena porque este es el tipo de valor devuelto del método. En cambio, aunque un nombre de espacio de nombres se clasifique como una expresión, no se evalúa como un valor y por tanto nunca puede ser el resultado final de una expresión. No puede pasar un nombre de espacio de nombres a un parámetro de método, usarlo en una nueva expresión ni asignarlo a una variable. Solo puede usarlo como una subexpresión en una expresión mayor. Lo mismo sucede con los tipos (a diferencia de los objetos System.Type), los nombres de grupo de métodos (a diferencia de los métodos específicos) y los descriptores de acceso add y remove de eventos.

Todos los valores tienen un tipo asociado. Por ejemplo, si x e y son variables de tipo `int`, el valor de la expresión `x + y` también es de tipo `int`. Si el valor se asigna a una variable de un tipo diferente, o si x e y son de tipos diferentes, se aplican las reglas de conversión de tipos. Para obtener más información sobre el funcionamiento de estas conversiones, vea Conversiones de tipos (Guía de programación de C#).

Desbordamientos

Las expresiones numéricas pueden producir desbordamientos si el valor es mayor que el valor máximo del tipo del valor. Para obtener más información, consulte las secciones Comprobadas y no comprobadas y Conversiones numéricas explícitas del artículo Conversiones numéricas integradas.

Prioridad y asociatividad de los operadores

Las reglas de asociatividad y prioridad de operadores rigen la manera en la que se evalúa una expresión. Para obtener más información, vea Operadores (Guía de programación de C#).

La mayoría de las expresiones, excepto las expresiones de asignación y las expresiones de invocación de método, se deben insertar en una instrucción. Para obtener más información, vea Instrucciones (Guía de programación de C#).

Literales y nombres simples

Los dos tipos de expresiones más simples son literales y nombres simples. Un literal es un valor constante que no tiene ningún nombre. Por ejemplo, en el código siguiente, `5` y `"Hello World"` son valores literales:

```
// Expression statements.
int i = 5;
string s = "Hello World";
```

Para obtener más información sobre literales, vea Tipos (Referencia de C#).

En el ejemplo anterior, tanto `i` como `s` son nombres simples que identifican variables locales. Cuando estas variables se usan en una expresión, el nombre de variable se evalúa como el valor almacenado actualmente en la ubicación de la variable en memoria. Esto se muestra en el ejemplo siguiente:

```
int num = 5;
System.Console.WriteLine(num); // Output: 5
num = 6;
System.Console.WriteLine(num); // Output: 6
```

Expresiones de invocación

En el ejemplo de código siguiente, la llamada a `DoWork` es una expresión de invocación.

```
DoWork();
```

Una invocación de método requiere el nombre del método, ya sea un nombre como en el ejemplo anterior o el resultado de otra expresión, seguido de paréntesis y los parámetros del método. Para más información, vea Métodos. Una invocación de delegado usa el nombre de un delegado y los parámetros de método entre paréntesis. Para obtener más información, vea Delegados (Guía de programación de C#). Las invocaciones de método e invocaciones de delegado se evalúan como el valor que devuelve el método, si devuelve alguno. Los métodos que devuelven un valor nulo no se pueden usar en lugar de un valor en una expresión.

Expresiones de consulta

A las expresiones de consulta se les aplican las mismas reglas que a las expresiones en general. Para más información, consulte LINQ.

Expresiones lambda

Las expresiones lambda representan "métodos insertados" que no tienen nombre pero pueden tener parámetros de entrada y varias instrucciones. Se usan mucho en LINQ para pasar argumentos a métodos. Las expresiones lambda se compilan en delegados o árboles de expresión, dependiendo del contexto en el que se usen. Para obtener más información, vea Expresiones lambda.

Árboles de expresión

Los árboles de expresión permiten representar las expresiones como estructuras de datos. Los proveedores LINQ los usan en gran medida para convertir expresiones de consulta en código significativo en otro contexto, como una base de datos SQL. Para obtener más información, vea Árboles de expresión (C#).

Definiciones de cuerpos de expresión

C# admite *miembros con forma de expresión*, que le permiten proporcionar una definición de cuerpo de expresión concisa para métodos, constructores, finalizadores, propiedades e indexadores. Para obtener más información, vea Miembros con forma de expresión.

Comentarios

Siempre que se identifica un acceso a una variable, propiedad de objeto o indexador de objeto desde una expresión, el valor de ese elemento se usa como valor de la expresión. Una expresión se puede colocar en cualquier parte de C# donde se requiera un valor u objeto, siempre que la expresión finalmente se evalúe como el tipo requerido.

especificación del lenguaje C#

Para más información, consulte la sección Expresiones de la especificación del lenguaje C#.

Vea también

- Guía de programación de C#
- Operadores
- Métodos
- Delegados
- Tipos
- LINQ

Miembros con cuerpo de expresión (Guía de programación de C#)

18/03/2020 • 6 minutes to read • Edit Online

Las definiciones de cuerpos de expresión permiten proporcionar la implementación de un miembro de una forma muy concisa y legible. Se puede usar una definición de cuerpo de expresión siempre que la lógica de cualquier miembro compatible, como un método o propiedad, se componga de una expresión única. Una definición de cuerpo de expresión tiene la siguiente sintaxis general:

```
member => expression;
```

donde *expresión* es una expresión válida.

La compatibilidad con las definiciones de cuerpos de expresión se introdujo para los métodos y las propiedades de solo lectura en C# 6 y se expandió en C# 7.0. Las definiciones de cuerpos de expresión se pueden usar con los miembros de tipo que se muestran en la tabla siguiente:

MEMBER	SE ADMITE DESDE...
Método	C# 6
Propiedad de solo lectura	C# 6
Propiedad	C# 7.0
Constructor	C# 7.0
Finalizador	C# 7.0
Indexador	C# 7.0

Métodos

Un método con cuerpo de expresión consta de una sola expresión que devuelve un valor cuyo tipo coincide con el tipo de valor devuelto del método, o bien, para los métodos que devuelven `void`, que realiza alguna operación. Por ejemplo, los tipos que reemplazan el método ToString normalmente incluyen una sola expresión que devuelve la representación de cadena del objeto actual.

En el ejemplo siguiente se define una clase `Person` que reemplaza el método ToString con una definición de cuerpo de expresión. También define un método `DisplayName` que muestra un nombre en la consola. Tenga en cuenta que la palabra clave `return` no se usa en la definición de cuerpo de expresión de `ToString`.

```
using System;

public class Person
{
   public Person(string firstName, string lastName)
   {
      fname = firstName;
      lname = lastName;
   }

   private string fname;
   private string lname;

   public override string ToString() => $"{fname} {lname}".Trim();
   public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
   static void Main()
   {
      Person p = new Person("Mandy", "Dejesus");
      Console.WriteLine(p);
      p.DisplayName();
   }
}
```

Para más información, vea Métodos (Guía de programación de C#).

Propiedades de solo lectura

A partir de C# 6, puede usar la definición de cuerpo de expresión para implementar una propiedad de solo lectura. Para ello, use la sintaxis siguiente:

```
PropertyType PropertyName => expression;
```

En el ejemplo siguiente se define una clase `Location` cuya propiedad `Name` de solo lectura se implementa como una definición de cuerpo de expresión que devuelve el valor del campo privado `locationName`:

```
public class Location
{
   private string locationName;

   public Location(string name)
   {
      locationName = name;
   }

   public string Name => locationName;
}
```

Para más información sobre las propiedades, vea Propiedades (Guía de programación de C#).

Propiedades

A partir de C# 7.0, puede usar las definiciones de cuerpo de expresión para implementar los descriptores de acceso `get` y `set` de propiedades. En el ejemplo siguiente se muestra cómo hacerlo:

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Para más información sobre las propiedades, vea Propiedades (Guía de programación de C#).

Constructores

Una definición de cuerpo de expresión para un constructor normalmente consta de una expresión de asignación única o una llamada de método que controla los argumentos del constructor o inicializa el estado de la instancia.

En el ejemplo siguiente se define una clase `Location` cuyo constructor tiene un único parámetro de cadena denominado *name*. La definición del cuerpo de expresión asigna el argumento a la propiedad `Name`.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Para más información, vea Constructores (Guía de programación de C#).

Finalizadores

Una definición de cuerpo de expresión para un finalizador normalmente contiene instrucciones de limpieza, como las instrucciones que liberan recursos no administrados.

En el ejemplo siguiente se define un finalizador que usa una definición de cuerpo de expresión para indicar que el finalizador se ha llamado.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

Para más información, vea Finalizadores (Guía de programación de C#).

Indizadores

Como las propiedades, los descriptores de acceso `get` y `set` del indizador constan de las definiciones de cuerpos de expresión si el descriptor de acceso `get` está formado por una sola expresión que devuelve un valor o el descriptor de acceso `set` realiza una asignación simple.

En el ejemplo siguiente se define una clase denominada `Sports` que incluye una matriz String interna que contiene los nombres de varios deportes. Los descriptores de acceso `get` y `set` del indizador se implementan como definiciones de cuerpos de expresión.

```
using System;
using System.Collections.Generic;

public class Sports
{
   private string[] types = { "Baseball", "Basketball", "Football",
                              "Hockey", "Soccer", "Tennis",
                              "Volleyball" };

   public string this[int i]
   {
      get => types[i];
      set => types[i] = value;
   }
}
```

Para más información, vea Indizadores (Guía de programación de C#).

Funciones anónimas (Guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

Una función anónima es una instrucción o expresión "alineada" que se puede usar siempre que se espera un tipo delegado. Se puede usar para inicializar un delegado con nombre o se puede pasar como un parámetro de método en lugar de un tipo delegado con nombre.

Puede usar una expresión lambda o un método anónimo para crear una función anónima. Se recomienda usar expresiones lambda porque proporcionan una manera más concisa y expresiva para escribir código alineado. A diferencia de los métodos anónimos, algunos tipos de expresiones lambda se pueden convertir en los tipos de árbol de expresión.

La evolución de los delegados en C#

En C# 1.0, una instancia de un delegado se creaba al inicializarla de forma explícita con un método que se definía en otro lugar en el código. C# 2.0 introdujo el concepto de métodos anónimos como una manera de escribir bloques de instrucciones insertados sin nombre que se podían ejecutar en la invocación de un delegado. C# 3.0 introdujo las expresiones lambda, que son similares en concepto a los métodos anónimos, pero más expresivas y concisas. Estas dos características se conocen colectivamente como *funciones anónimas*. Por lo general, las aplicaciones que tienen como destino .NET Framework 3.5 y versiones posteriores deberían usar expresiones lambda.

En el ejemplo siguiente se muestra la evolución de la creación de delegados desde C# 1.0 a C# 3.0:

```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Hello. My name is M and I write lines.
    That's nothing. I'm anonymous and
    I'm a famous author.
    Press any key to exit.
 */
```

Especificación del lenguaje C#

Para obtener más información, vea la sección Expresiones de función anónima de la Especificación del lenguaje C#.

Vea también

- Instrucciones, expresiones y operadores
- Expresiones lambda
- Delegados
- Árboles de expresión (C#)

Expresiones lambda (Guía de programación de C#)

29/06/2020 • 20 minutes to read • Edit Online

Una *expresión lambda* es una expresión que tiene cualquiera de estas dos formas:

- Una lambda de expresión que tiene una expresión como cuerpo:

```
(input-parameters) => expression
```

- Una lambda de instrucción que tiene un bloque de instrucciones como cuerpo:

```
(input-parameters) => { <sequence-of-statements> }
```

Use el operador de declaración lambda `=>` para separar la lista de parámetros de la lamba de su cuerpo. Para crear una expresión lambda, especifique los parámetros de entrada (si existen) a la izquierda del operador lambda y una expresión o bloque de instrucciones en el otro lado.

Toda expresión lambda se puede convertir en un tipo delegado. El tipo delegado al que se puede convertir una expresión lambda se define según los tipos de sus parámetros y el valor devuelto. Si una expresión lambda no devuelve un valor, se puede convertir en uno de los tipos delegados `Action`; de lo contrario, se puede convertir en uno de los tipos delegados `Func`. Por ejemplo, una expresión lambda que tiene dos parámetros y no devuelve ningún valor corresponde a un delegado Action<T1,T2>. Una expresión lambda que tiene un parámetro y devuelve un valor se puede convertir en un delegado Func<T,TResult>. En el ejemplo siguiente, la expresión lambda `x => x * x`, que especifica un parámetro denominado `x` y devuelve el valor de `x` al cuadrado, se asigna a una variable de un tipo delegado:

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Las expresiones lambda también se pueden convertir en los tipos de árbol de expresión, como se muestra en los ejemplos siguientes:

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

Puede usar expresiones lambda en cualquier código que requiera instancias de tipos delegados o de árboles de expresión, por ejemplo, como un argumento del método Task.Run(Action) para pasar el código que se debe ejecutar en segundo plano. También puede usar expresiones lambda al escribir LINQ en C#, como se muestra en el ejemplo siguiente:

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

Cuando se usa la sintaxis de método para llamar al método Enumerable.Select en la clase System.Linq.Enumerable, por ejemplo en LINQ to Objects y en LINQ to XML, el parámetro es un tipo delegado System.Func<T,TResult>. Cuando se llama al método Queryable.Select en la clase System.Linq.Queryable, por ejemplo en LINQ to SQL, el tipo de parámetro es un tipo de árbol de expresión `Expression<Func<TSource,TResult>>`. En ambos casos, se puede usar la misma expresión lambda para especificar el valor del parámetro. Esto hace que las dos llamadas `Select` tengan un aspecto similar aunque, de hecho, el tipo de objetos creados a partir las lambdas es distinto.

Lambdas de expresión

Una expresión lambda con una expresión en el lado derecho del operador `=>` se denomina *lambda de expresión*. Las lambdas de expresión se usan ampliamente en la construcción de árboles de expresión. Una expresión lambda devuelve el resultado de evaluar la condición y tiene la siguiente forma:

```
(input-parameters) => expression
```

Los paréntesis solo son opcionales si la lambda tiene un parámetro de entrada; de lo contrario, son obligatorios.

Para especificar cero parámetros de entrada, utilice paréntesis vacíos:

```
Action line = () => Console.WriteLine();
```

Dos o más parámetros de entrada se separan por comas y se encierran entre paréntesis:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

A veces resulta imposible que el compilador deduzca los tipos de entrada. Puede especificar los tipos de manera explícita, tal como se muestra en el ejemplo siguiente:

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Los tipos de parámetro de entrada deben ser todos explícitos o todos implícitos; de lo contrario, se produce un error del compilador CS0748.

El cuerpo de una expresión lambda puede constar de una llamada al método. Sin embargo, si crea árboles de expresión que se evalúan fuera del contexto de Common Language Runtime de .NET, como en SQL Server, no debe utilizar llamadas a métodos en expresiones lambda. Los métodos no tendrán ningún significado fuera del contexto de .NET Common Language Runtime.

Lambdas de instrucción

Una lambda de instrucción es similar a un lambda de expresión, salvo que las instrucciones se encierran entre llaves:

```
(input-parameters) => { <sequence-of-statements> }
```

El cuerpo de una lambda de instrucción puede estar compuesto de cualquier número de instrucciones; sin embargo, en la práctica, generalmente no hay más de dos o tres.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

Las expresiones lambdas no se pueden usar para crear árboles de expresión.

Lambdas asincrónicas

Puede crear fácilmente expresiones e instrucciones lambda que incorporen el procesamiento asincrónico mediante las palabras clave async y await . Por ejemplo, en el siguiente ejemplo de formularios Windows Forms se incluye un controlador de eventos que llama y espera un método asincrónico, `ExampleMethodAsync`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Puede agregar el mismo controlador de eventos utilizando una lambda asincrónica. Para agregar este controlador, agregue un modificador `async` antes de la lista de parámetros lambda, como se muestra en el ejemplo siguiente:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Para obtener más información sobre cómo crear y usar métodos asincrónicos, vea Programación asincrónica con async y await.

Expresiones lambda y tuplas

A partir de C# 7.0, el lenguaje C# proporciona compatibilidad integrada para las tuplas. Puede proporcionar una tupla como argumento a una expresión lambda, mientras que la expresión lambda también puede devolver una tupla. En algunos casos, el compilador de C# usa la inferencia de tipos para determinar los tipos de componentes de la tupla.

Para definir una tupla, incluya entre paréntesis una lista delimitada por comas de los componentes. En el ejemplo siguiente se usa la tupla con tres componentes para pasar una secuencia de números a una expresión lambda, que duplica cada valor y devuelve una tupla con tres componentes que contiene el resultado de las multiplicaciones.

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 *
ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

Normalmente, los campos de una tupla se denominan `Item1`, `Item2`, etc., aunque puede definir una tupla con componentes con nombre, como en el ejemplo siguiente.

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 *
ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

Para más información sobre las tuplas de C#, consulte el artículo sobre los tipos de tuplas de C#.

Lambdas con los operadores de consulta estándar

LINQ to Objects, entre otras implementaciones, tiene un parámetro de entrada cuyo tipo es uno de la familia Func<TResult> de delegados genéricos. Estos delegados usan parámetros de tipo para definir el número y el tipo de los parámetros de entrada, así como el tipo de valor devuelto del delegado. Los

delegados `Func` son muy útiles para encapsular expresiones definidas por el usuario que se aplican a cada elemento en un conjunto de datos de origen. Por ejemplo, considere el tipo delegado Func<T,TResult>:

```
public delegate TResult Func<in T, out TResult>(T arg)
```

Se pueden crear instancias del delegado como una instancia `Func<int, bool>`, donde `int` es un parámetro de entrada y `bool` es el valor devuelto. El valor devuelto siempre se especifica en el último parámetro de tipo. Por ejemplo, `Func<int, string, bool>` define un delegado con dos parámetros de entrada, `int` y `string`, y un tipo de valor devuelto de `bool`. El delegado `Func` siguiente, cuando se invoca, devuelve un valor booleano que indica si el parámetro de entrada es igual a cinco:

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

También puede proporcionar una expresión lambda cuando el tipo de argumento es Expression<TDelegate>, (por ejemplo, en los operadores de consulta estándar que se definen en el tipo Queryable). Al especificar un argumento Expression<TDelegate>, la lambda se compila en un árbol de expresión.

En el ejemplo siguiente se usa el operador de consulta estándar Count:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

El compilador puede deducir el tipo del parámetro de entrada o también se puede especificar explícitamente. Esta expresión lambda particular cuenta aquellos enteros (`n`) que divididos por dos dan como resto 1.

El siguiente ejemplo genera una secuencia que contiene todos los elementos de la matriz `numbers` que preceden al 9, ya que ese es el primer número de la secuencia que no cumple la condición:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

En el siguiente ejemplo se especifican varios parámetros de entrada encerrándolos entre paréntesis. El método devuelve todos los elementos de la matriz `numbers` hasta que encuentra un número cuyo valor es menor que la posición ordinal en la matriz:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

Inferencia de tipos en expresiones lambda

Al escribir lambdas, no tiene por qué especificar un tipo para los parámetros de entrada, ya que el

compilador puede deducir el tipo según el cuerpo de lambda, los tipos de parámetros y otros factores, tal como se describe en la especificación del lenguaje C#. Para la mayoría de los operadores de consulta estándar, la primera entrada es el tipo de los elementos en la secuencia de origen. Si está realizando una consulta sobre `IEnumerable<Customer>`, se deducirá que la variable de entrada será un objeto `Customer`, lo cual significa que se podrá tener acceso a sus métodos y propiedades:

```
customers.Where(c => c.City == "London");
```

Las reglas generales para la inferencia de tipos de las lambdas son las siguientes:

- La lambda debe contener el mismo número de parámetros que el tipo delegado.
- Cada parámetro de entrada de la lambda debe poder convertirse implícitamente a su parámetro de delegado correspondiente.
- El valor devuelto de la lambda (si existe) debe poder convertirse implícitamente al tipo de valor devuelto del delegado.

Observe que las expresiones lambda, en sí mismas, no tienen tipo, ya que el sistema de tipos comunes no tiene ningún concepto intrínseco de "expresión lambda". Sin embargo, a veces resulta práctico hablar coloquialmente del "tipo" de una expresión lambda. En estos casos, el tipo hace referencia al tipo del delegado o el tipo de Expression en el que se convierte la expresión lambda.

Captura de variables externas y el ámbito de las variables en las expresiones lambda

Las operaciones lambda pueden hacer referencia a *variables externas*. Estas son las variables que están en el ámbito del método que define la expresión lambda o en el ámbito del tipo que contiene la expresión lambda. Las variables que se capturan de esta manera se almacenan para su uso en la expresión lambda, cuando de otro modo quedarían fuera de ámbito y serían eliminadas por la recolección de elementos no utilizados. Para poder utilizar una variable externa en una expresión lambda, debe estar previamente asignada. En el ejemplo siguiente se muestran estas reglas:

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {j}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}: {result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable:
{equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

Las reglas siguientes se aplican al ámbito de las variables en las expresiones lambda:

- Una variable capturada no se recolectará como elemento no utilizado hasta que el delegado que hace referencia a ella sea elegible para la recolección de elementos no utilizados.
- Las variables introducidas en una expresión lambda no son visibles en el método envolvente.
- Una expresión lambda no puede capturar directamente un parámetro in, ref ni out desde el método envolvente.
- Una instrucción return en una expresión lambda no hace que el método envolvente devuelva un valor.

- Una expresión lambda no puede contener una instrucción goto, break ni continue si el destino de esa instrucción de salto está fuera del bloque de la expresión lambda. También es un error utilizar una instrucción de salto fuera del bloque de la expresión lambda si el destino está dentro del bloque.

Especificación del lenguaje C#

Para obtener más información, vea la sección Expresiones de función anónima de la Especificación del lenguaje C#.

Capítulo destacado del libro

Delegates, Events, and Lambda Expressions (Delegados, eventos y expresiones lambda) en C# 3.0 Cookbook, Tercera edición: More than 250 solutions for C# 3.0 programmers (Más de 250 soluciones para programadores de C# 3.0)

Vea también

- Guía de programación de C#
- LINQ (Language Integrated Query)
- Árboles de expresión
- Funciones locales frente a expresiones lambda
- Ejemplos de C# de Visual Studio 2008 (vea los archivos de ejemplos de consultas LINQ y el programa XQuery)
- Expresiones lambda recursivas

Procedimiento Usar expresiones lambda en una consulta (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

No se pueden usar expresiones lambda directamente en la sintaxis de consulta, pero sí en llamadas de método, y las expresiones de consulta pueden contener llamadas de método. De hecho, algunas operaciones de consulta solo se pueden expresar con la sintaxis de método. Para obtener más información sobre la diferencia entre la sintaxis de consulta y la sintaxis de método, vea Sintaxis de consultas y sintaxis de métodos en LINQ.

Ejemplo

En el siguiente ejemplo se muestra cómo usar una expresión lambda en una consulta basada en métodos con el operador de consulta estándar Enumerable.Where. Tenga en cuenta que el método Where de este ejemplo tiene un parámetro de entrada de tipo delegado Func<T,TResult> y que el delegado toma un entero como entrada y devuelve un valor booleano. La expresión lambda se puede convertir en ese delegado. Si se tratara de una consulta de LINQ to SQL que usa el método Queryable.Where, el tipo de parámetro sería `Expression<Func<int,bool>>` pero la expresión lambda tendría exactamente el mismo aspecto. Para más información sobre el tipo Expression, vea System.Linq.Expressions.Expression.

```
class SimpleLambda
{
    static void Main()
    {

        // Data source.
        int[] scores = { 90, 71, 82, 93, 75, 82 };

        // The call to Count forces iteration of the source
        int highScoreCount = scores.Where(n => n > 80).Count();

        Console.WriteLine("{0} scores are greater than 80", highScoreCount);

        // Outputs: 4 scores are greater than 80
    }
}
```

Ejemplo

En el ejemplo siguiente se muestra cómo usar una expresión lambda en una llamada de método de una expresión de consulta. La expresión lambda es necesaria porque el operador de consulta estándar Sum no se puede invocar con la sintaxis de consulta.

La consulta agrupa primero los alumnos por curso, tal y como se define en la enumeración `GradeLevel`. Luego, suma las puntuaciones totales por alumno de cada grupo. Esto requiere dos operaciones `Sum`. La `Sum` interna calcula la puntuación total de cada alumno y la `Sum` externa mantiene un total acumulado conjunto de todos los alumnos del grupo.

```
private static void TotalsByGradeLevel()
{
    // This query retrieves the total scores for First Year students, Second Years, and so on.
    // The outer Sum method uses a lambda in order to specify which numbers to add together.
    var categories =
    from student in students
    group student by student.Year into studentGroup
    select new { GradeLevel = studentGroup.Key, TotalScore = studentGroup.Sum(s => s.ExamScores.Sum()) };

    // Execute the query.
    foreach (var cat in categories)
    {
        Console.WriteLine("Key = {0} Sum = {1}", cat.GradeLevel, cat.TotalScore);
    }
}
/*
     Outputs:
     Key = SecondYear Sum = 1014
     Key = ThirdYear Sum = 964
     Key = FirstYear Sum = 1058
     Key = FourthYear Sum = 974
*/
```

Compilar el código

Para ejecutar este código, copie y pegue el método en la clase `StudentClass` que se especifica en Consultar una colección de objetos y llámelo desde el método `Main`.

Vea también

- Expresiones lambda
- Árboles de expresión (C#)

Comparaciones de igualdad (guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

A veces es necesario comparar si dos valores son iguales. En algunos casos, se prueba la *igualdad de valores*, también denominada *equivalencia*, lo que significa que los valores contenidos en las dos variables son iguales. En otros casos, hay que determinar si dos variables hacen referencia al mismo objeto subyacente de la memoria. Este tipo de igualdad se denomina *igualdad de referencia* o *identidad*. En este tema se describen estos dos tipos de igualdad y se proporcionan vínculos a otros temas para obtener más información.

Igualdad de referencia

La igualdad de referencia significa que dos referencias de objeto hacen referencia al mismo objeto subyacente. Esto puede suceder mediante una asignación simple, como se muestra en el ejemplo siguiente.

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

En este código, se crean dos objetos, pero después de la instrucción de asignación, ambas referencias hacen referencia al mismo objeto. Por consiguiente, presentan igualdad de referencia. Use el método ReferenceEquals para determinar si dos referencias hacen referencia al mismo objeto.

El concepto de igualdad de la referencia solo se aplica a los tipos de referencia. Los objetos de tipo de valor no pueden presentar igualdad de referencia porque al asignar una instancia de un tipo de valor a una variable, se realiza una copia del valor. Por consiguiente, nunca puede haber dos structs con conversión unboxing que hagan referencia a la misma ubicación de la memoria. Además, si se usa ReferenceEquals para comparar dos tipos de valor, el resultado siempre será `false`, aunque todos los valores que contengan los objetos sean idénticos. Esto se debe a que a cada variable se aplica la conversión boxing en una instancia de objeto independiente. Para más

información, consulte Procedimiento Probar la igualdad de referencia (Identidad).

Igualdad de valores

La igualdad de valores significa que dos objetos contienen el mismo valor o valores. Para los tipos de valor primitivos, como int o bool, las pruebas de igualdad de valores son sencillas. Puede usar el operador `==`, como se muestra en el ejemplo siguiente.

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

Para la mayoría de los otros tipos, las pruebas de igualdad de valores son más complejas, porque es preciso entender cómo la define el tipo. Para las clases y los structs que tienen varios campos o propiedades, la igualdad de valores suele definirse de modo que significa que todos los campos o propiedades tienen el mismo valor. Por ejemplo, podrían definirse dos objetos `Point` que fueran equivalentes si pointA.X es igual a pointB.X y pointA.Y es igual a pointB.Y.

En cambio, no hay ningún requisito que exija que la equivalencia se base en todos los campos de un tipo. Se puede basar en un subconjunto. Al comparar tipos que no sean de su propiedad, es importante asegurarse concretamente de cómo se define la equivalencia para ese tipo. Para más información sobre cómo definir la igualdad de valores en sus propias clases y structs, consulte Procedimiento Definir la igualdad de valores para un tipo.

Igualdad de valores en valores de número de punto flotante

Las comparaciones de igualdad de valores de punto flotante (double y float) son problemáticas debido a la imprecisión de la aritmética de número de punto flotante en los equipos binarios. Para obtener más información, vea los comentarios en el tema System.Double.

Temas relacionados

TITLE	DESCRIPCIÓN
Procedimiento Probar la igualdad de referencia (Identidad)	Describe cómo determinar si dos variables presentan igualdad de referencia.
Procedimiento Definir la igualdad de valores para un tipo	Describe cómo proporcionar una definición personalizada de igualdad de valores para un tipo.
Guía de programación de C#	Proporciona vínculos a información detallada sobre importantes características del lenguaje C# y características que están disponibles para C# a través de .NET.
Tipos	Proporciona información sobre el sistema de tipos de C# y vínculos a información adicional.

Vea también

- Guía de programación de C#

Procedimiento: Definición de la igualdad de valores para un tipo (Guía de programación de C#)

18/03/2020 • 12 minutes to read • Edit Online

Cuando defina una clase o un struct, debe decidir si tiene sentido crear una definición personalizada de igualdad (o equivalencia) de valores para el tipo. Normalmente, la igualdad de valores se implementa cuando se espera agregar objetos del tipo a una colección de algún tipo, o cuando su objetivo principal es almacenar un conjunto de campos o propiedades. Puede basar la definición de la igualdad de valores en una comparación de todos los campos y propiedades del tipo, o bien puede basarla en un subconjunto.

En cualquier caso, tanto en las clases como en las estructuras, la implementación debe cumplir las cinco garantías de equivalencia (en las siguientes reglas, se da por hecho que `x`, `y` y `z` no son NULL):

1. `x.Equals(x)` devuelve `true`. Esto se denomina propiedad reflexiva.
2. `x.Equals(y)` devuelve el mismo valor que `y.Equals(x)`. Esto se denomina propiedad simétrica.
3. Si `(x.Equals(y) && y.Equals(z))` devuelve `true`, `x.Equals(z)` devuelve `true`. Esto se denomina propiedad transitiva.
4. Las invocaciones sucesivas de `x.Equals(y)` devuelven el mismo valor siempre y cuando los objetos a los que x e y hacen referencia no se modifiquen.
5. Cualquier valor distinto de NULL no es igual a NULL. Pero CLR comprueba si hay valores NULL en todas las llamadas a métodos y produce una `NullReferenceException` si la referencia `this` fuera NULL. Por lo tanto, `x.Equals(y)` produce una excepción cuando `x` es NULL. Esto rompe las reglas 1 o 2, en función del argumento de `Equals`.

Cualquier struct que defina ya tiene una implementación predeterminada de igualdad de valor que hereda de la invalidación System.ValueType del método Object.Equals(Object). Esta implementación usa la reflexión para examinar todos los campos y propiedades del tipo. Aunque esta implementación genera resultados correctos, es relativamente lenta en comparación con una implementación personalizada escrita específicamente para el tipo.

Los detalles de implementación para la igualdad de valores son diferentes para las clases y los structs. A pesar de ello, tanto las clases como los structs requieren los mismos pasos básicos para implementar la igualdad:

1. Invalide el método virtual Object.Equals(Object). En la mayoría de los casos, la implementación de `bool Equals( object obj )` debería llamar solamente al método `Equals` específico del tipo que es la implementación de la interfaz System.IEquatable<T>. (Vea el paso 2).
2. Implemente la interfaz System.IEquatable<T> proporcionando un método `Equals` específico del tipo. Aquí es donde se realiza la comparación de equivalencias propiamente dicha. Por ejemplo, podría decidir que, para definir la igualdad, solo se comparen uno o dos campos del tipo. No genere excepciones desde `Equals`. Solo para las clases: este método debe examinar únicamente los campos que se declaran en la clase. Debe llamar a `base.Equals` para examinar los campos que están en la clase base. (No realice esto si el tipo hereda directamente de Object, porque la implementación Object de Object.Equals(Object) realiza una comprobación de igualdad de referencia).
3. Opcional pero recomendado: Sobrecargue los operadores == y !=.
4. Invalide Object.GetHashCode de manera que dos objetos que tengan igualdad de valor produzcan el mismo código hash.

5. Opcional: Para admitir definiciones para "mayor que" o "menor que", implemente la interfaz IComparable<T> para el tipo y sobrecargue los operadores <= y >=.

En el primer ejemplo que aparece más abajo se muestra una implementación de clase. En el segundo ejemplo se muestra una implementación de struct.

Ejemplo

En el ejemplo siguiente se muestra cómo implementar la igualdad de valores en una clase (tipo de referencia).

```
namespace ValueEquality
{
    using System;
    class TwoDPoint : IEquatable<TwoDPoint>
    {
        // Readonly auto-implemented properties.
        public int X { get; private set; }
        public int Y { get; private set; }

        // Set the properties in the constructor.
        public TwoDPoint(int x, int y)
        {
            if ((x < 1) || (x > 2000) || (y < 1) || (y > 2000))
            {
                throw new System.ArgumentException("Point must be in range 1 - 2000");
            }
            this.X = x;
            this.Y = y;
        }

        public override bool Equals(object obj)
        {
            return this.Equals(obj as TwoDPoint);
        }

        public bool Equals(TwoDPoint p)
        {
            // If parameter is null, return false.
            if (Object.ReferenceEquals(p, null))
            {
                return false;
            }

            // Optimization for a common success case.
            if (Object.ReferenceEquals(this, p))
            {
                return true;
            }

            // If run-time types are not exactly the same, return false.
            if (this.GetType() != p.GetType())
            {
                return false;
            }

            // Return true if the fields match.
            // Note that the base class is not invoked because it is
            // System.Object, which defines Equals as reference equality.
            return (X == p.X) && (Y == p.Y);
        }

        public override int GetHashCode()
        {
            return X * 0x00010000 + Y;
        }
```

```
    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        // Check for null on left side.
        if (Object.ReferenceEquals(lhs, null))
        {
            if (Object.ReferenceEquals(rhs, null))
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
    {
        return !(lhs == rhs);
    }
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new System.ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj)
    {
        return this.Equals(obj as ThreeDPoint);
    }

    public bool Equals(ThreeDPoint p)
    {
        // If parameter is null, return false.
        if (Object.ReferenceEquals(p, null))
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
```

```
        }

        public override int GetHashCode()
        {
            return (X * 0x100000) + (Y * 0x1000) + Z;
        }

        public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
        {
            // Check for null.
            if (Object.ReferenceEquals(lhs, null))
            {
                if (Object.ReferenceEquals(rhs, null))
                {
                    // null == null = true.
                    return true;
                }

                // Only the left side is null.
                return false;
            }
            // Equals handles the case of null on right side.
            return lhs.Equals(rhs);
        }

        public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs)
        {
            return !(lhs == rhs);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
            ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
            ThreeDPoint pointC = null;
            int i = 5;

            Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
            Console.WriteLine("Compare to some other type = {0}", pointA.Equals(i));

            TwoDPoint pointD = null;
            TwoDPoint pointE = null;

            Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD == pointE);

            pointE = new TwoDPoint(3, 4);
            Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
            Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
            Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

            System.Collections.ArrayList list = new System.Collections.ArrayList();
            list.Add(new ThreeDPoint(3, 4, 5));
            Console.WriteLine("pointE.Equals(list[0]): {0}", pointE.Equals(list[0]));

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        null comparison = False
```

```
        null comparison = False
        Compare to some other type = False
        Two null TwoDPoints are equal: True
        (pointE == pointA) = False
        (pointA == pointE) = False
        (pointA != pointE) = True
        pointE.Equals(list[0]): False
    */
}
```

En las clases (tipos de referencia), la implementación predeterminada de ambos métodos Object.Equals(Object) realiza una comparación de igualdad de referencia, no una comprobación de igualdad de valores. Cuando un implementador invalida el método virtual, lo hace para asignarle semántica de igualdad de valores.

Los operadores `==` y `!=` pueden usarse con clases, incluso si la clase no los sobrecarga, pero el comportamiento predeterminado consiste en realizar una comprobación de igualdad de referencia. En una clase, si sobrecarga el método `Equals`, debería sobrecargar los operadores `==` y `!=`, pero no es obligatorio.

Ejemplo

En el ejemplo siguiente se muestra cómo implementar la igualdad de valores en un struct (tipo de valor):

```
using System;
struct TwoDPoint : IEquatable<TwoDPoint>
{
    // Read/write auto-implemented properties.
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
        : this()
    {
        X = x;
        Y = x;
    }

    public override bool Equals(object obj)
    {
        if (obj is TwoDPoint)
        {
            return this.Equals((TwoDPoint)obj);
        }
        return false;
    }

    public bool Equals(TwoDPoint p)
    {
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode()
    {
        return X ^ Y;
    }

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
    {
        return !(lhs.Equals(rhs));
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        TwoDPoint pointA = new TwoDPoint(3, 4);
        TwoDPoint pointB = new TwoDPoint(3, 4);
        int i = 5;

        // Compare using virtual Equals, static Equals, and == and != operators.
        // True:
        Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
        // True:
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        // True:
        Console.WriteLine("object.Equals(pointA, pointB) = {0}", object.Equals(pointA, pointB));
        // False:
        Console.WriteLine("pointA.Equals(null) = {0}", pointA.Equals(null));
        // False:
        Console.WriteLine("(pointA == null) = {0}", pointA == null);
        // True:
        Console.WriteLine("(pointA != null) = {0}", pointA != null);
        // False:
        Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
        // CS0019:
        // Console.WriteLine("pointA == i = {0}", pointA == i);

        // Compare unboxed to boxed.
        System.Collections.ArrayList list = new System.Collections.ArrayList();
        list.Add(new TwoDPoint(3, 4));
        // True:
        Console.WriteLine("pointA.Equals(list[0]): {0}", pointA.Equals(list[0]));

        // Compare nullable to nullable and to non-nullable.
        TwoDPoint? pointC = null;
        TwoDPoint? pointD = null;
        // False:
        Console.WriteLine("pointA == (pointC = null) = {0}", pointA == pointC);
        // True:
        Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

        TwoDPoint temp = new TwoDPoint(3, 4);
        pointC = temp;
        // True:
        Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA == pointC);

        pointD = temp;
        // True:
        Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD == pointC);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    Object.Equals(pointA, pointB) = True
    pointA.Equals(null) = False
    (pointA == null) = False
    (pointA != null) = True
    pointA.Equals(i) = False
    pointE.Equals(list[0]): True
    pointA == (pointC = null) = False
    pointC == pointD = True
    pointA == (pointC = 3,4) = True
    pointD == (pointC = 3,4) = True
```

```
    */
}
```

Para los structs, la implementación predeterminada de Object.Equals(Object) (que es la versión invalidada de System.ValueType) realiza una comprobación de igualdad de valor con la reflexión para comparar valores de cada campo en el tipo. Cuando un implementador invalida el método `Equals` virtual en un struct, lo hace para proporcionar un medio más eficaz de llevar a cabo la comprobación de igualdad de valores y, opcionalmente, para basar la comparación en un subconjunto de propiedades o campos del struct.

Los operadores `==` y `!=` no pueden funcionar en un struct a menos que el struct los sobrecargue explícitamente.

Vea también

- Comparaciones de igualdad
- Guía de programación de C#

Procedimiento Probar la igualdad de referencias (identidad) (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

No tiene que implementar ninguna lógica personalizada para admitir las comparaciones de igualdad de referencias en los tipos. Esta funcionalidad se proporciona para todos los tipos mediante el método Object.ReferenceEquals estático.

En el ejemplo siguiente, se muestra cómo determinar si dos variables tienen *igualdad de referencia*, lo que significa que hacen referencia al mismo objeto en la memoria.

En el ejemplo también se muestra por qué Object.ReferenceEquals siempre devuelve `false` para los tipos de valor y por qué no se debe usar ReferenceEquals para determinar la igualdad entre cadenas.

Ejemplo

```
namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After asignment: ReferenceEquals(tcA, tcB) = {0}",
                              Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
```

```
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
            Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name, tcB.Num);
            #endregion

            // Demonstrate that two value type instances never have reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are copied on assignment. tsD and tsC have
            // the same values but are not the same object.
            TestStruct tsD = tsC;
            Console.WriteLine("After asignment: ReferenceEquals(tsC, tsD) = {0}",
                                Object.ReferenceEquals(tsC, tsD)); // false
            #endregion

            #region stringRefEquality
            // Constant strings within the same assembly are always interned by the runtime.
            // This means they are stored in the same location in memory. Therefore,
            // the two strings have reference equality although no assignment takes place.
            string strA = "Hello world!";
            string strB = "Hello world!";
            Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                                Object.ReferenceEquals(strA, strB)); // true

            // After a new string is assigned to strA, strA and strB
            // are no longer interned and no longer have reference equality.
            strA = "Goodbye world!";
            Console.WriteLine("strA = \"{0}\" strB = \"{1}\"", strA, strB);

            Console.WriteLine("After strA changes, ReferenceEquals(strA, strB) = {0}",
                                Object.ReferenceEquals(strA, strB)); // false

            // A string that is created at runtime cannot be interned.
            StringBuilder sb = new StringBuilder("Hello world!");
            string stringC = sb.ToString();
            // False:
            Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                                Object.ReferenceEquals(stringC, strB));

            // The string class overloads the == operator to perform an equality comparison.
            Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

            #endregion

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    ReferenceEquals(tcA, tcB) = False
    After asignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After asignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
*/
```

La implementación de `Equals` en la clase base universal System.Object también realiza una comprobación de la igualdad de referencias; sin embargo, es mejor no seguir este procedimiento porque, en el caso de que una clase invalide el método, los resultados podrían no ser los esperados. Lo mismo se cumple para los operadores `==` y `!=`. Cuando se usan en tipos de referencia, el comportamiento predeterminado de `==` y `!=` consiste en realizar

una comprobación de la igualdad de referencias. Sin embargo, las clases derivadas pueden sobrecargar el operador para realizar una comprobación de la igualdad de valores. Para minimizar las posibilidades de error, lo mejor es usar siempre ReferenceEquals cuando deba determinarse si dos objetos tienen igualdad de referencia.

El runtime siempre aplica el método Intern a las cadenas constantes dentro del mismo ensamblado. Es decir, solo se conserva una instancia de cada cadena literal única. Sin embargo, el runtime no garantiza que se vaya a aplicar el método Intern a las cadenas creadas en tiempo de ejecución, ni tampoco que dicho método se aplique a dos cadenas constantes iguales en distintos ensamblados.

Vea también

- Comparaciones de igualdad

Tipos (Guía de programación de C#)

29/06/2020 • 23 minutes to read • Edit Online

Tipos, variables y valores

C# es un lenguaje fuertemente tipado. Todas las variables y constantes tienen un tipo, al igual que todas las expresiones que se evalúan como un valor. Cada declaración del método especifica un nombre, un número de parámetros, un tipo y una naturaleza (valor, referencia o salida) para cada parámetro de entrada y para el valor devuelto. La biblioteca de clases .NET define un conjunto de tipos numéricos integrados, así como tipos más complejos que representan una amplia variedad de construcciones lógicas, como el sistema de archivos, conexiones de red, colecciones y matrices de objetos, y fechas. Los programas de C# típicos usan tipos de la biblioteca de clases, así como tipos definidos por el usuario que modelan los conceptos que son específicos del dominio del problema del programa.

Entre la información almacenada en un tipo se puede incluir lo siguiente:

- El espacio de almacenamiento que requiere una variable del tipo.
- Los valores máximo y mínimo que puede representar.
- Los miembros (métodos, campos, eventos, etc.) que contiene.
- El tipo base del que hereda.
- La ubicación donde se asignará la memoria para variables en tiempo de ejecución.
- Los tipos de operaciones permitidas.

El compilador usa información de tipo para garantizar que todas las operaciones que se realizan en el código cuentan *con seguridad de tipos*. Por ejemplo, si declara una variable de tipo int, el compilador le permite usar la variable en operaciones de suma y resta. Si intenta realizar esas mismas operaciones en una variable de tipo bool, el compilador genera un error, como se muestra en el siguiente ejemplo:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
> Los desarrolladores de C y C++ deben tener en cuenta que, en C#, bool no se puede convertir en int.

El compilador inserta la información de tipo en el archivo ejecutable como metadatos. Common Language Runtime (CLR) usa esos metadatos en tiempo de ejecución para garantizar aún más la seguridad de tipos cuando asigna y reclama memoria.

Especificar tipos en declaraciones de variable

Cuando declare una variable o constante en un programa, debe especificar su tipo o usar la palabra clave var para que el compilador infiera el tipo. En el ejemplo siguiente se muestran algunas declaraciones de variable que utilizan tanto tipos numéricos integrados como tipos complejos definidos por el usuario:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

Los tipos de parámetros del método y los valores devueltos se especifican en la declaración del método. En la siguiente firma se muestra un método que requiere una variable int como argumento de entrada y devuelve una cadena:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

Tras declarar una variable, no se puede volver a declarar con un nuevo tipo y no se le puede asignar un valor que no sea compatible con su tipo declarado. Por ejemplo, no puede declarar un valor int y, luego, asignarle un valor booleano de `true`. En cambio, los valores se pueden convertir en otros tipos, por ejemplo, cuando se asignan a variables nuevas o se pasan como argumentos de método. El compilador realiza automáticamente una *conversión de tipo* que no da lugar a una pérdida de datos. Una conversión que pueda dar lugar a la pérdida de datos requiere un valor *cast* en el código fuente.

Para obtener más información, vea Conversiones de tipos.

Tipos integrados

C# proporciona un conjunto estándar de tipos numéricos integrados para representar números enteros, valores de punto flotante, expresiones booleanas, caracteres de texto, valores decimales y otros tipos de datos. También hay tipos `string` y `object` integrados. Están disponibles para su uso en cualquier programa de C#. Para obtener una lista completa de los tipos integrados, vea Tipos integrados.

Tipos personalizados

Las construcciones struct, class, interface y enum se utilizan para crear sus propios tipos personalizados. La biblioteca de clases .NET es en sí misma una colección de tipos personalizados proporcionados por Microsoft que puede usar en sus propias aplicaciones. De forma predeterminada, los tipos usados con más frecuencia en la biblioteca de clases están disponibles en cualquier programa de C#. Otros están disponibles solo cuando agrega explícitamente una referencia de proyecto al ensamblado en el que se definen. Una vez que el compilador tenga una referencia al ensamblado, puede declarar variables (y constantes) de los tipos declarados en dicho ensamblado en el código fuente. Para más información, vea Biblioteca de clases .NET.

Common Type System

Es importante entender dos aspectos fundamentales sobre el sistema de tipos en .NET:

- Es compatible con el principio de herencia. Los tipos pueden derivarse de otros tipos, denominados *tipos base*. El tipo derivado hereda (con algunas restricciones), los métodos, las propiedades y otros miembros del tipo base. A su vez, el tipo base puede derivarse de algún otro tipo, en cuyo caso el tipo derivado hereda los miembros de ambos tipos base en su jerarquía de herencia. Todos los tipos, incluidos los tipos numéricos integrados como System.Int32 (palabra clave de C#: int), derivan en última instancia de un único tipo base, que es System.Object (palabra clave de C#: object). Esta jerarquía de tipos unificada se denomina Common Type System (CTS). Para más información sobre la herencia en C#, vea Herencia.

- En CTS, cada tipo se define como un *tipo de valor* o un *tipo de referencia*. Esto incluye todos los tipos personalizados de la biblioteca de clases .NET y también sus propios tipos definidos por el usuario. Los tipos que se definen mediante el uso de la palabra clave struct son tipos de valor; todos los tipos numéricos integrados son `structs`. Los tipos que se definen mediante el uso de la palabra clave class son tipos de referencia. Los tipos de referencia y los tipos de valor tienen distintas reglas de tiempo de compilación y distintos comportamientos de tiempo de ejecución.

En la ilustración siguiente se muestra la relación entre los tipos de valor y los tipos de referencia en CTS.

En la imagen siguiente se muestran tipos de valores y tipos de referencias en CTS:

> **NOTE**
> Puede ver que los tipos utilizados con mayor frecuencia están organizados en el espacio de nombres System. Sin embargo, el espacio de nombres que contiene un tipo no tiene ninguna relación con un tipo de valor o un tipo de referencia.

Tipos de valor

Los tipos de valor derivan de System.ValueType, el cual deriva de System.Object. Los tipos que derivan de System.ValueType tienen un comportamiento especial en CLR. Las variables de tipo de valor contienen directamente sus valores, lo que significa que la memoria se asigna insertada en cualquier contexto en el que se declare la variable. No se produce ninguna asignación del montón independiente ni sobrecarga de la recolección de elementos no utilizados para las variables de tipo de valor.

Existen dos categorías de tipos de valor: struct y enum.

Los tipos numéricos integrados son structs y tienen campos y métodos a los que se puede acceder:

```
// constant field on type byte.
byte b = byte.MaxValue;
```

Pero se declaran y se les asignan valores como si fueran tipos simples no agregados:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Los tipos de valor están *sellados*, lo que significa que, por ejemplo, no puede derivar un tipo de System.Int32, y no puede definir un struct para que herede de cualquier clase o struct definido por el usuario, porque un struct solo puede heredar de System.ValueType. A pesar de ello, un struct puede implementar una o más interfaces. Puede convertir un tipo struct en cualquier tipo de interfaz que implemente. Esto hace que una operación de conversión *boxing* encapsule el struct dentro de un objeto de tipo de referencia en el montón administrado. Las operaciones de conversión boxing se producen cuando se pasa un tipo de valor a un método que toma System.Object o cualquier tipo de interfaz como parámetro de entrada. Para obtener más información, vea Conversión boxing y unboxing.

Puede usar la palabra clave struct para crear sus propios tipos de valor personalizados. Normalmente, un struct se usa como un contenedor para un pequeño conjunto de variables relacionadas, como se muestra en el ejemplo siguiente:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

Para más información sobre estructuras, vea Tipos de estructura. Para más información sobre los tipos de valor, vea Tipos de valor.

La otra categoría de tipos de valor es enum. Una enumeración define un conjunto de constantes integrales con nombre. Por ejemplo, la enumeración System.IO.FileMode de la biblioteca de clases .NET contiene un conjunto de enteros constantes con nombre que especifican cómo se debe abrir un archivo. Se define como se muestra en el ejemplo siguiente:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

La constante `System.IO.FileMode.Create` tiene un valor de 2. Sin embargo, el nombre es mucho más significativo para los humanos que leen el código fuente y, por esa razón, es mejor utilizar enumeraciones en lugar de números literales constantes. Para obtener más información, vea System.IO.FileMode.

Todas las enumeraciones se heredan de System.Enum, el cual se hereda de System.ValueType. Todas las reglas

que se aplican a las estructuras también se aplican a las enumeraciones. Para más información sobre las enumeraciones, vea Tipos de enumeración.

Tipos de referencia

Un tipo que se define como clase, delegado, matriz o interfaz es un *tipo de referencia*. Al declarar una variable de un tipo de referencia en tiempo de ejecución, esta contendrá el valor null hasta que se cree explícitamente un objeto mediante el operador new, o bien que se le asigne un objeto creado en otro lugar mediante `new`, tal y como se muestra en el ejemplo siguiente:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

Una interfaz debe inicializarse junto con un objeto de clase que lo implementa. Si `MyClass` implementa `IMyInterface`, cree una instancia de `IMyInterface`, tal como se muestra en el ejemplo siguiente:

```
IMyInterface iface = new MyClass();
```

Cuando se crea el objeto, se asigna la memoria en el montón administrado y la variable solo contiene una referencia a la ubicación del objeto. Los tipos del montón administrado producen sobrecarga cuando se asignan y cuando los reclama la función de administración de memoria automática de CLR, conocida como *recolección de elementos no utilizados*. En cambio, la recolección de elementos no utilizados también está muy optimizada y no crea problemas de rendimiento en la mayoría de los escenarios. Para obtener más información sobre la recolección de elementos no utilizados, vea Administración de memoria automática.

Todas las matrices son tipos de referencia, incluso si sus elementos son tipos de valor. Las matrices derivan de manera implícita de la clase System.Array, pero el usuario las declara y las usa con la sintaxis simplificada que proporciona C#, como se muestra en el ejemplo siguiente:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Los tipos de referencia admiten la herencia completamente. Al crear una clase, puede heredar de cualquier otra interfaz o clase que no esté definida como sealed; y otras clases pueden heredar de la clase e invalidar sus métodos virtuales. Para más información sobre cómo crear clases propias, vea Clases y structs. Para más información sobre la herencia y los métodos virtuales, vea Herencia.

Tipos de valores literales

En C#, los valores literales reciben un tipo del compilador. Puede especificar cómo debe escribirse un literal numérico; para ello, anexe una letra al final del número. Por ejemplo, para especificar que el valor 4.56 debe tratarse como un valor flotante, anexe "f" o "F" después del número: `4.56f`. Si no se anexa ninguna letra, el compilador inferirá un tipo para el literal. Para obtener más información sobre los tipos que se pueden especificar con sufijos de letras, vea Tipos numéricos integrales y Tipos numéricos de punto flotante.

Dado que los literales tienen tipo y todos los tipos derivan en última instancia de System.Object, puede escribir y compilar código como el siguiente:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Tipos genéricos

Los tipos se pueden declarar con uno o varios *parámetros de tipo* que actúan como un marcador de posición para el tipo real (el *tipo concreto*) que proporcionará el código de cliente cuando cree una instancia del tipo. Estos tipos se denominan *tipos genéricos*. Por ejemplo, el tipo de .NET System.Collections.Generic.List<T> tiene un parámetro de tipo al que, por convención, se le denomina *T*. Cuando crea una instancia del tipo, especifica el tipo de los objetos que contendrá la lista, por ejemplo, la cadena:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

El uso del parámetro de tipo permite reutilizar la misma clase para incluir cualquier tipo de elemento, sin necesidad de convertir cada elemento en object. Las clases de colección genéricas se denominan *colecciones fuertemente tipadas* porque el compilador conoce el tipo específico de los elementos de la colección y puede generar un error en tiempo de compilación si, por ejemplo, intenta agregar un valor entero al objeto `stringList` del ejemplo anterior. Para más información, vea Genéricos.

Tipos implícitos, tipos anónimos y tipos que admiten un valor NULL

Como se ha mencionado anteriormente, puede asignar implícitamente un tipo a una variable local (pero no miembros de clase) mediante la palabra clave var. La variable sigue recibiendo un tipo en tiempo de compilación, pero este lo proporciona el compilador. Para más información, vea Variables locales con asignación implícita de tipos.

En algunos casos, resulta conveniente crear un tipo con nombre para conjuntos sencillos de valores relacionados que no desea almacenar ni pasar fuera de los límites del método. Puede crear *tipos anónimos* para este fin. Para obtener más información, consulte Tipos anónimos (Guía de programación de C#).

Los tipos de valor normales no pueden tener un valor null, pero se pueden crear tipos de valor que aceptan valores NULL mediante la adición de `?` después del tipo. Por ejemplo, `int?` es un tipo `int` que también puede tener el valor null. Los tipos que admiten un valor NULL son instancias del tipo struct genérico System.Nullable<T>. Los tipos que admiten un valor NULL son especialmente útiles cuando hay un intercambio de datos con bases de datos en las que los valores numéricos podrían ser nulos. Para más información, vea Tipos que admiten un valor NULL.

Secciones relacionadas

Para obtener más información, vea los temas siguientes:

- Conversiones de tipos
- Conversión boxing y conversión unboxing
- Uso de tipo dinámico

- Tipos de valor
- Tipos de referencia
- Clases y structs
- Tipos anónimos
- Genéricos

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Conversión de tipos de datos XML
- Tipos enteros

Conversiones de tipos (Guía de programación de C#)

16/04/2020 • 9 minutes to read • Edit Online

Dado que C# tiene tipos estáticos en tiempo de compilación, después de declarar una variable, no se puede volver a declarar ni se le puede asignar un valor de otro tipo a menos que ese tipo sea convertible de forma implícita al tipo de la variable. Por ejemplo, `string` no se puede convertir de forma implícita a `int`. Por tanto, después de declarar `i` como un valor `int`, no se le puede asignar la cadena "Hello", como se muestra en el código siguiente:

```
int i;
i = "Hello"; // error CS0029: Cannot implicitly convert type 'string' to 'int'
```

Pero es posible que en ocasiones sea necesario copiar un valor en una variable o parámetro de método de otro tipo. Por ejemplo, es posible que tenga una variable de entero que se necesita pasar a un método cuyo parámetro es de tipo `double`. O es posible que tenga que asignar una variable de clase a una variable de tipo de interfaz. Estos tipos de operaciones se denominan *conversiones de tipos*. En C#, se pueden realizar las siguientes conversiones de tipos:

- **Conversiones implícitas**: no se requiere ninguna sintaxis especial porque la conversión tiene seguridad de tipos y no se perderá ningún dato. Los ejemplos incluyen conversiones de tipos enteros más pequeños a más grandes, y conversiones de clases derivadas a clases base.
- **Conversiones explícitas**: las conversiones explícitas requieren una expresión Cast. La conversión es necesaria si es posible que se pierda información en la conversión, o cuando es posible que la conversión no sea correcta por otros motivos. Entre los ejemplos típicos están la conversión numérica a un tipo que tiene menos precisión o un intervalo más pequeño, y la conversión de una instancia de clase base a una clase derivada.
- **Conversiones definidas por el usuario**: las conversiones definidas por el usuario se realizan por medio de métodos especiales que se pueden definir para habilitar las conversiones explícitas e implícitas entre tipos personalizados que no tienen una relación de clase base-clase derivada. Para obtener más información, vea Operadores de conversión definidos por el usuario.
- **Conversiones con clases del asistente**: para realizar conversiones entre tipos no compatibles, como enteros y objetos System.DateTime, o cadenas hexadecimales y matrices de bytes puede usar la clase System.BitConverter, la clase System.Convert y los métodos `Parse` de los tipos numéricos integrados, como Int32.Parse. Para obtener más información, consulte Procedimiento Convertir una matriz de bytes en un valor int, Procedimiento Convertir una cadena en un número y Procedimiento Convertir cadenas hexadecimales en tipos numéricos.

Conversiones implícitas

Para los tipos numéricos integrados, se puede realizar una conversión implícita cuando el valor que se va a almacenar se puede encajar en la variable sin truncarse ni redondearse. Para los tipos enteros, esto significa que el intervalo del tipo de origen es un subconjunto apropiado del intervalo para el tipo de destino. Por ejemplo, una variable de tipo long (entero de 64 bits) puede almacenar cualquier valor que un tipo int (entero de 32 bits) pueda almacenar. En el ejemplo siguiente, el compilador convierte de forma implícita el valor de `num` en la parte derecha a un tipo `long` antes de asignarlo a `bigNum`.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

Para obtener una lista completa de las conversiones numéricas implícitas, consulte la sección Conversiones numéricas implícitas del artículo Conversiones numéricas integradas.

Para los tipos de referencia, siempre existe una conversión implícita desde una clase a cualquiera de sus interfaces o clases base directas o indirectas. No se necesita ninguna sintaxis especial porque una clase derivada siempre contiene a todos los miembros de una clase base.

```
Derived d = new Derived();
Base b = d; // Always OK.
```

Conversiones explícitas

Pero si no se puede realizar una conversión sin riesgo de perder información, el compilador requiere que se realice una conversión explícita, que se denomina *conversión*. Una conversión de tipos es una manera de informar explícitamente al compilador de que se pretende realizar la conversión y se es consciente de que se puede producir pérdida de datos. Para realizar una conversión, especifique el tipo al que se va a convertir entre paréntesis delante del valor o la variable que se va a convertir. El siguiente programa convierte un tipo double en un tipo int. El programa no se compilará sin la conversión.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

Para obtener una lista completa de las conversiones numéricas explícitas admitidas, consulte la sección Conversiones numéricas explícitas del artículo Conversiones numéricas integradas.

Para los tipos de referencia, se requiere una conversión explícita si es necesario convertir de un tipo base a un tipo derivado:

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe) a;
```

Una operación de conversión entre tipos de referencia no cambia el tipo en tiempo de ejecución del objeto

subyacente. Solo cambia el tipo del valor que se usa como referencia a ese objeto. Para obtener más información, vea Polimorfismo .

Excepciones de conversión de tipos en tiempo de ejecución

En algunas conversiones de tipos de referencia, el compilador no puede determinar si una conversión será válida. Es posible que una operación de conversión que se compile correctamente produzca un error en tiempo de ejecución. Como se muestra en el ejemplo siguiente, una conversión de tipo que produce un error en tiempo de ejecución produce una excepción InvalidCastException.

```
using System;

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // Cause InvalidCastException at run time
        // because Mammal is not convertible to Reptile.
        Reptile r = (Reptile)a;
    }
}
```

C# proporciona el operador is para permitir probar la compatibilidad antes de realizar una conversión. Para obtener más información, consulte Procedimiento para convertir de forma segura mediante la coincidencia de patrones y los operadores is y as.

Especificación del lenguaje C#

Para obtener más información, vea la sección sobre conversiones de la Especificación del lenguaje C#.

Vea también

- Guía de programación de C#
- Tipos
- Expresión Cast
- Operadores de conversión definidos por el usuario
- Conversión de tipos generalizada
- Procedimiento Convertir una cadena en un número

Conversión boxing y unboxing (Guía de programación de C#)

20/05/2020 • 9 minutes to read • Edit Online

La conversión boxing es el proceso de convertir un tipo de valor en el tipo `object` o en cualquier tipo de interfaz implementado por este tipo de valor. Cuando Common Language Runtime (CLR) aplica la conversión boxing a un tipo de valor, ajusta el valor dentro de una instancia System.Object y lo almacena en el montón administrado. La conversión unboxing extrae el tipo de valor del objeto. La conversión boxing es implícita y la conversión unboxing es explícita. El concepto de conversión boxing y unboxing es la base de la vista unificada del sistema de tipos de C#, en el que un valor de cualquier tipo se puede tratar como objeto.

En el ejemplo siguiente, se aplica *conversión boxing* a la variable de entero `i` y esta se asigna al objeto `o`.

```
int i = 123;
// The following line boxes i.
object o = i;
```

Luego se puede aplicar conversión unboxing al objeto `o` y asignarlo a la variable de entero `i`:

```
o = 123;
i = (int)o;  // unboxing
```

En los ejemplos siguientes se muestra cómo usar la conversión boxing en C#.

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
```

```
// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

Rendimiento

Con relación a las asignaciones simples, las conversiones boxing y unboxing son procesos que consumen muchos recursos. Cuando se aplica la conversión boxing a un tipo de valor, se debe asignar y construir un objeto completamente nuevo. En menor grado, la conversión de tipos requerida para aplicar la conversión unboxing también es costosa. Para más información, vea Rendimiento.

Boxing

La conversión boxing se utiliza para almacenar tipos de valor en el montón de recolección de elementos no utilizados. La conversión boxing es una conversión implícita de un tipo de valor en el tipo `object` o en cualquier tipo de interfaz implementado por este tipo de valor. Al aplicar la conversión boxing a un tipo de valor se asigna una instancia de objeto en el montón y se copia el valor en el nuevo objeto.

Considere la siguiente declaración de una variable de tipo de valor:

```
int i = 123;
```

La siguiente instrucción aplica implícitamente la operación de conversión boxing en la variable `i`:

```
// Boxing copies the value of i into object o.
object o = i;
```

El resultado de esta instrucción es crear una referencia de objeto `o` en la pila que hace referencia a un valor del tipo `int` en el montón. Este valor es una copia del tipo de valor asignado a la variable `i`. La diferencia entre las dos variables, `i` y `o`, se muestra en la imagen siguiente de la conversión boxing:

También es posible realizar la conversión boxing de manera explícita, tal como se muestra en el ejemplo siguiente, pero esta nunca es necesaria:

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Description

Este ejemplo convierte una variable de entero `i` en un objeto `o` mediante la conversión boxing. A continuación, el valor almacenado en la variable `i` se cambia de `123` a `456`. El ejemplo muestra que el tipo de valor original y el objeto al que se ha aplicado la conversión boxing usan ubicaciones de memoria independientes y, por consiguiente, pueden almacenar valores diferentes.

Ejemplo

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

Conversión unboxing

La conversión unboxing es una conversión explícita del tipo `object` en un tipo de valor o de un tipo de interfaz en un tipo de valor que implementa la interfaz. Una operación de conversión unboxing consiste en lo siguiente:

- Comprobar la instancia de objeto para asegurarse de que se trata de un valor de conversión boxing del tipo de valor dado.
- Copiar el valor de la instancia en la variable de tipo de valor.

Las siguientes instrucciones muestran las operaciones de conversión boxing y unboxing:

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

En la figura siguiente se muestra el resultado de las instrucciones anteriores:

Para que la conversión unboxing de tipos de valor sea correcta en tiempo de ejecución, el elemento al que se aplica debe ser una referencia a un objeto creado previamente mediante la conversión boxing de una instancia de ese tipo de valor. Si se intenta aplicar la conversión unboxing a `null`, se producirá una excepción NullReferenceException. Si se intenta aplicar la conversión unboxing a una referencia de un tipo de valor incompatible, se producirá una excepción InvalidCastException.

Ejemplo

El ejemplo siguiente muestra un caso de conversión unboxing no válida y la excepción `InvalidCastException` resultante. Si se utiliza `try` y `catch`, se muestra un mensaje de error cuando se produce el error.

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

Este programa produce el resultado siguiente:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

Si cambia la instrucción:

```
int j = (short) o;
```

a:

```
int j = (int) o;
```

la conversión se realizará y se obtendrá el resultado:

```
Unboxing OK.
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Tipos de referencia
- Tipos de valor

Procedimiento Convertir una matriz de bytes en un valor int (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo usar la clase BitConverter para convertir una matriz de bytes en un valor int y de nuevo en una matriz de bytes. Por ejemplo, es posible que tenga que realizar una conversión de bytes a un tipo de datos integrado después de leer los bytes fuera de la red. Además del método ToInt32(Byte[], Int32) del ejemplo, en la tabla siguiente se muestran los métodos de la clase BitConverter que sirven para convertir bytes (de una matriz de bytes) en otros tipos integrados.

TIPO DEVUELTO	MÉTODO
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

Ejemplo

En este ejemplo se inicializa una matriz de bytes, se invierte la matriz si la arquitectura de equipo es little-endian (es decir, en primer lugar se almacena el byte menos significativo) y, después, se llama al método ToInt32(Byte[], Int32) para convertir cuatro bytes de la matriz en `int`. El segundo argumento de ToInt32(Byte[], Int32) especifica el índice de inicio de la matriz de bytes.

> **NOTE**
>
> El resultado puede cambiar en función de los modos endian de la arquitectura del equipo.

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

Ejemplo

En este ejemplo, el método GetBytes(Int32) de la clase BitConverter se llama para convertir `int` en una matriz de bytes.

> **NOTE**
>
> El resultado puede cambiar en función de los modos endian de la arquitectura del equipo.

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

Vea también

- BitConverter
- IsLittleEndian
- Tipos

Procedimiento Convertir una cadena en un número (Guía de programación de C#)

29/06/2020 • 7 minutes to read • Edit Online

Puede convertir una cadena en un número llamando al método `Parse` o `TryParse` que se encuentra en varios tipos numéricos (`int`, `long`, `double`, etc.) o usando métodos en la clase System.Convert.

Si tiene una cadena, resulta algo más eficaz y sencillo llamar a un método `TryParse` (por ejemplo, `int.TryParse("11", out number)`) o un método `Parse` (por ejemplo, `var number = int.Parse("11")`). El uso de un método Convert resulta más práctico para objetos generales que implementan IConvertible.

Puede usar los métodos `Parse` o `TryParse` sobre el tipo numérico que espera que la cadena contenga, como el tipo System.Int32. El método Convert.ToInt32 utiliza Parse internamente. El método `Parse` devuelve el número convertido; el método `TryParse` devuelve un valor Boolean que indica si la conversión se realizó correctamente, y devuelve el número convertido en un parámetro `out`. Si el formato de la cadena no es válido, `Parse` produce una excepción, mientras que `TryParse` devuelve `false`. Cuando se llama a un método `Parse`, siempre debe usar control de excepciones para detectar un FormatException en caso de que se produzca un error en la operación de análisis.

Llamada a los métodos Parse y TryParse

Los métodos `Parse` y `TryParse` no tienen en cuenta los espacios en blanco al principio y al final de la cadena, pero todos los demás caracteres deben ser caracteres que formen el tipo numérico adecuado (`int`, `long`, `ulong`, `float`, `decimal`, etc.). Si hay un espacio en blanco dentro de la cadena que forma el número, se producirá un error. Por ejemplo, puede usar `decimal.TryParse` para analizar "10", "10,3" o " 10 ", pero no para analizar 10 en "10X", "1 0" (advierta el espacio insertado), "10 ,3" (advierta el espacio insertado) o "10e1" (`float.TryParse` funciona aquí), y así sucesivamente. Además, una cadena cuyo valor es `null` o String.Empty no se puede analizar correctamente. Puede buscar una cadena nula o vacía antes de intentar analizarla mediante una llamada al método String.IsNullOrEmpty.

En el siguiente ejemplo se muestran llamadas correctas e incorrectas a `Parse` y `TryParse`.

```
using System;

public class StringConversion
{
    public static void Main()
    {
       string input = String.Empty;
       try
       {
           int result = Int32.Parse(input);
           Console.WriteLine(result);
       }
       catch (FormatException)
       {
           Console.WriteLine($"Unable to parse '{input}'");
       }
       // Output: Unable to parse ''

       try
       {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
       }
       catch (FormatException e)
       {
           Console.WriteLine(e.Message);
       }
       // Output: -105

        if (Int32.TryParse("-105", out int j))
            Console.WriteLine(j);
        else
            Console.WriteLine("String could not be parsed.");
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
            Console.WriteLine(inputString);
        else
            Console.WriteLine($"Int32.TryParse could not parse '{inputString}' to an int.");
        // Output: Int32.TryParse could not parse 'abc' to an int.
     }
}
```

El ejemplo siguiente muestra una un enfoque para analizar una cadena que se espera que incluya caracteres numéricos iniciales (incluidos caracteres hexadecimales) y caracteres no numéricos finales. Asigna caracteres válidos desde el principio de una cadena a una nueva cadena antes de llamar al método TryParse. Dado que las cadenas que va a analizar contienen un pequeño número de caracteres, el ejemplo llama al método String.Concat para asignar caracteres válidos a una nueva cadena. Para una cadena más larga, se puede usar la clase StringBuilder en su lugar.

```
using System;

public class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = String.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || (Char.ToUpperInvariant(c) >= 'A' && Char.ToUpperInvariant(c) <=
'F') || c == ' ') {
                numericString = String.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }
        if (int.TryParse(numericString, System.Globalization.NumberStyles.HexNumber, null, out int i))
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
            // Output: '  10FFxxx' --> '  10FF' --> 4351

        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str) {
            // Check for numeric characters (0-9), a negative sign, or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = String.Concat(numericString, c);
            } else
            {
                break;
            }
        }
        if (int.TryParse(numericString, out int j))
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
            // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Llamada a los métodos Convert

En la siguiente tabla se muestran algunos de los métodos de la clase Convert que se pueden usar para convertir una cadena en un número.

TIPO NUMÉRICO	MÉTODO
`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)

TIPO NUMÉRICO	MÉTODO
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

En este ejemplo, se llama al método Convert.ToInt32(String) para convertir una cadena de entrada en un valor int. El ejemplo detecta las dos excepciones más comunes que este método puede generar, FormatException y OverflowException. Si se puede incrementar el número resultante sin exceder Int32.MaxValue, el ejemplo suma 1 al resultado y muestra la salida.

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): ");

            string input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond its current value");
                }
           }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string go = Console.ReadLine();
            if (go.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): -1000
//   The new value is -999
//   Go again? Y/N: n
```

Vea también

- Tipos
- Determinación de si una cadena representa un valor numérico
- Ejemplo: Utilidad de formato WinForms de .NET Core (C#)

Procedimiento Convertir cadenas hexadecimales en tipos numéricos (Guía de programación de C#)

18/03/2020 • 5 minutes to read • Edit Online

En estos ejemplos se muestra cómo realizar las tareas siguientes:

- Obtener el valor hexadecimal de cada uno de los caracteres de un elemento string.
- Obtener el elemento char que corresponde a cada valor de una cadena hexadecimal.
- Convertir un elemento `string` hexadecimal en un elemento int.
- Convertir un elemento `string` hexadecimal en un elemento float.
- Convertir una matriz de bytes en un elemento `string` hexadecimal.

Ejemplo

En este ejemplo se genera el valor hexadecimal de cada uno de los caracteres de `string`. Primero, analiza `string` como una matriz de caracteres. Después, llama a ToInt32(Char) en cada carácter para obtener su valor numérico. Finalmente, aplica formato al número como su representación hexadecimal en un elemento `string`.

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

Ejemplo

En este ejemplo se analiza un elemento `string` de valores hexadecimales y genera el carácter correspondiente a cada valor hexadecimal. Primero, llama al método Split(Char[]) para obtener cada valor hexadecimal como un elemento `string` individual en una matriz. Después, llama a ToInt32(String, Int32) para convertir el valor hexadecimal a un valor decimal representado como int. Muestra dos maneras distintas de obtener el carácter correspondiente a ese código de carácter. En la primera técnica se usa ConvertFromUtf32(Int32), que devuelve el carácter correspondiente al argumento de tipo entero como `string`. En la segunda técnica, `int` se convierte de

manera explícita en un elemento char.

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value = {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

Ejemplo

En este ejemplo se muestra otra manera de convertir un `string` hexadecimal en un entero mediante la llamada al método Parse(String, NumberStyles).

```
string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

Ejemplo

En el siguiente ejemplo se muestra cómo convertir un `string` hexadecimal en un elemento float con la clase System.BitConverter y el método UInt32.Parse.

```
string hexString = "43480170";
uint num = uint.Parse(hexString, System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

Ejemplo

En el ejemplo siguiente se muestra cómo convertir una matriz byte en una cadena hexadecimal mediante la clase System.BitConverter.

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

Vea también

- Cadenas con formato numérico estándar
- Tipos
- Determinación de si una cadena representa un valor numérico

Uso de tipo dinámico (Guía de programación de C#)

29/06/2020 • 9 minutes to read • Edit Online

C# 4 introduce un nuevo tipo, `dynamic`. Se trata de un tipo estático, pero un objeto de tipo `dynamic` omite la comprobación de tipos estáticos. En la mayoría de los casos, funciona como si tuviera el tipo `object`. En tiempo de compilación, se supone que un elemento con tipo `dynamic` admite cualquier operación. Por consiguiente, no tendrá que preocuparse de si el objeto obtiene su valor de una API de COM, de un lenguaje dinámico como IronPython, del Document Object Model (DOM) HTML, de la reflexión o de otro lugar en el programa. Pero si el código no es válido, los errores se detectan en tiempo de ejecución.

Por ejemplo, si la instancia de método `exampleMethod1` del código siguiente solo tiene un parámetro, el compilador reconoce que la primera llamada al método, `ec.exampleMethod1(10, 4)`, no es válida porque contiene dos argumentos. La llamada genera un error del compilador. El compilador no comprueba la segunda llamada al método, `dynamic_ec.exampleMethod1(10, 4)`, porque el tipo de `dynamic_ec` es `dynamic`. Por consiguiente, no se notifica ningún error del compilador. Pero el error no pasa inadvertido indefinidamente. Se detecta en tiempo de ejecución y genera una excepción en tiempo de ejecución.

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

El rol del compilador en estos ejemplos consiste en empaquetar información sobre lo que cada instrucción se propone para el objeto o la expresión con el tipo `dynamic`. En tiempo de ejecución, la información almacenada se examina y cualquier instrucción que no sea válida genera una excepción en tiempo de ejecución.

El resultado de la mayoría de las operaciones dinámicas es `dynamic`. Por ejemplo, si se sitúa el puntero del mouse sobre el uso de `testSum` en el ejemplo siguiente, IntelliSense muestra el tipo **(local variable) dynamic testSum**.

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Las operaciones en las que el resultado no es `dynamic` incluyen:

- Conversiones de `dynamic` a otro tipo.
- Llamadas de constructor que incluyen argumentos de tipo `dynamic`.

Por ejemplo, el tipo de `testInstance` en la declaración siguiente es `ExampleClass`, no `dynamic`:

```
var testInstance = new ExampleClass(d);
```

En la siguiente sección, "Conversiones", se muestran ejemplos de conversión.

Conversiones

Las conversiones entre objetos dinámicos y de otro tipo son fáciles. Esto permite al desarrollador cambiar entre el comportamiento dinámico y no dinámico.

Cualquier objeto se puede convertir implícitamente al tipo dinámico, tal y como se muestra en los ejemplos siguientes.

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

En cambio, una conversión implícita se puede aplicar dinámicamente a cualquier expresión de tipo `dynamic`.

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Resolución de sobrecarga con argumentos de tipo dinámico

La resolución de sobrecarga se produce en tiempo de ejecución y no en tiempo de compilación si uno o varios argumentos de una llamada de método tienen el tipo `dynamic` o si el receptor de la llamada de método es de tipo `dynamic`. En el ejemplo siguiente, si el único método `exampleMethod2` accesible se define para que tome un argumento de cadena, al enviar `d1` como argumento no se produce un error del compilador, pero sí se genera una excepción en tiempo de ejecución. La resolución de sobrecarga no se produce en tiempo de ejecución porque el tipo en tiempo de ejecución de `d1` es `int`, y `exampleMethod2` requiere una cadena.

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec is not
// dynamic. A run-time exception is raised because the run-time type of d1 is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Dynamic Language Runtime

Dynamic Language Runtime (DLR) es una API introducida en .NET Framework 4. Proporciona la infraestructura que admite el tipo `dynamic` en C# y también la implementación de lenguajes de programación dinámicos como IronPython e IronRuby. Para obtener más información sobre el entorno DLR, vea Información general acerca de Dynamic Language Runtime.

Interoperabilidad COM

C# 4 incluye varias características que mejoran la experiencia de interoperar con API de COM tales como las API de automatización de Office. Entre las mejoras se incluye el uso del tipo `dynamic` y de argumentos opcionales y con nombre.

Muchos métodos COM permiten variaciones en los tipos de argumento y el tipo de valor devuelto mediante la designación de los tipos como `object`. Esto requería una conversión explícita de los valores para coordinarlos con variables fuertemente tipadas en C#. Si se compila con la opción -link (opciones del compilador de C#), la introducción del tipo `dynamic` le permite tratar las repeticiones de `object` en las firmas COM como si fueran de tipo `dynamic` y así evitar gran parte de la conversión. Por ejemplo, las instrucciones siguientes comparan cómo se accede a una celda de una hoja de cálculo de Microsoft Office Excel con el tipo `dynamic` y sin el tipo `dynamic`.

```
// Before the introduction of dynamic.
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and
// the conversion to Excel.Range are handled by the run-time COM binder.
excelApp.Cells[1, 1].Value = "Name";
Excel.Range range2010 = excelApp.Cells[1, 1];
```

Temas relacionados

TITLE	DESCRIPCIÓN
dynamic	Describe el uso de la palabra clave `dynamic`.
Información general sobre Dynamic Language Runtime	Ofrece información general sobre DLR, que es un entorno en tiempo de ejecución que agrega un conjunto de servicios para lenguajes dinámicos en Common Language Runtime (CLR).
Tutorial: Crear y usar objetos dinámicos	Ofrece instrucciones paso a paso para crear un objeto dinámico personalizado y para crear un proyecto que acceda a una biblioteca de `IronPython`.
Procedimiento para acceder a objetos de interoperabilidad de Office mediante características de C#	Muestra cómo crear un proyecto que use argumentos opcionales y con nombre, el tipo `dynamic` y otras mejoras que simplifican el acceso a objetos de la API de Office.

Tutorial: Crear y usar objetos dinámicos (C# y Visual Basic)

18/03/2020 • 19 minutes to read • Edit Online

Los objetos dinámicos exponen miembros como propiedades y métodos en tiempo de ejecución, en lugar de en tiempo de compilación. Esto le permite crear objetos para trabajar con estructuras que no coinciden con un formato o tipo estático. Por ejemplo, puede usar un objeto dinámico para hacer referencia a Document Object Model (DOM) HTML, que puede contener cualquier combinación de atributos y elementos de marcado HTML válidos. Dado que cada documento HTML es único, los miembros de un documento HTML específico se determinan en tiempo de ejecución. Un método común para hacer referencia a un atributo de un elemento HTML consiste en pasar el nombre del atributo al método `GetProperty` del elemento. Para hacer referencia al atributo `id` del elemento HTML `<div id="Div1">`, primero debe obtener una referencia al elemento `<div>` y, después, usar `divElement.GetProperty("id")`. Si usa un objeto dinámico, puede hacer referencia al atributo `id` como `divElement.id`.

Los objetos dinámicos también proporcionan un acceso cómodo a lenguajes dinámicos como IronPython e IronRuby. Puede usar un objeto dinámico para hacer referencia a un script dinámico que se interpreta en tiempo de ejecución.

Para hacer referencia a un objeto dinámico, use un enlace en tiempo de ejecución. En C#, el tipo de un objeto enlazado en tiempo de ejecución se especifica como `dynamic`. En Visual Basic, el tipo de un objeto enlazado en tiempo de ejecución se especifica como `Object`. Para obtener más información, vea dynamic y Enlace en tiempo de compilación y en tiempo de ejecución.

Puede crear objetos dinámicos personalizados con las clases del espacio de nombres System.Dynamic. Por ejemplo, puede crear un objeto ExpandoObject y especificar los miembros de ese objeto en tiempo de ejecución. También puede crear su propio tipo que hereda la clase DynamicObject. Después, puede invalidar los miembros de la clase DynamicObject para proporcionar funciones dinámicas en tiempo de ejecución.

En este tutorial realizará las tareas siguientes:

- Crear un objeto personalizado que expone dinámicamente el contenido de un archivo de texto como propiedades de un objeto.
- Crear un proyecto que usa una biblioteca `IronPython`.

Requisitos previos

Necesita IronPython para .NET para poder seguir este tutorial. Vaya a su página de descarga para obtener la versión más reciente.

> **NOTE**
>
> Es posible que el equipo muestre nombres o ubicaciones diferentes para algunos de los elementos de la interfaz de usuario de Visual Studio en las siguientes instrucciones. La edición de Visual Studio que se tenga y la configuración que se utilice determinan estos elementos. Para obtener más información, vea Personalizar el IDE.

Crear un objeto dinámico personalizado

El primer proyecto que se crea en este tutorial define un objeto dinámico personalizado que busca en el contenido

de un archivo de texto. El texto que se va a buscar se especifica mediante el nombre de una propiedad dinámica. Por ejemplo, si el código de llamada especifica `dynamicFile.Sample`, la clase dinámica devuelve una lista genérica de cadenas que contiene todas las líneas del archivo que comienzan con "Sample". La búsqueda no distingue entre mayúsculas y minúsculas. La clase dinámica también admite dos argumentos opcionales. El primer argumento es un valor de enumeración de opción de búsqueda que especifica que la clase dinámica debe buscar coincidencias al principio de la línea, al final de la línea o en cualquier parte de la línea. El segundo argumento especifica que la clase dinámica debe recortar los espacios iniciales y finales de cada línea antes de buscar. Por ejemplo, si el código de llamada especifica `dynamicFile.Sample(StringSearchOption.Contains)`, la clase dinámica busca "Sample" en cualquier parte de una línea. Si el código de llamada especifica `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, la clase dinámica busca "Sample" al principio de cada línea y no quita los espacios iniciales y finales. El comportamiento predeterminado de la clase dinámica es buscar una coincidencia al principio de cada línea y quitar los espacios iniciales y finales.

Para crear una clase dinámica personalizada

1. Inicie Visual Studio.

2. En el menú **Archivo** , elija **Nuevo** y haga clic en **Proyecto**.

3. En el panel **Tipos de proyecto** del cuadro de diálogo **Nuevo proyecto**, asegúrese de que esté seleccionado **Windows**. Seleccione **Aplicación de consola** en el panel **Plantillas**. En el cuadro **Nombre**, escriba `DynamicSample` y haga clic en **Aceptar**. Se crea el proyecto.

4. Haga clic con el botón derecho en el proyecto DynamicSample, seleccione **Agregar** y, después, haga clic en **Clase**. En el cuadro **Nombre**, escriba `ReadOnlyFile` y haga clic en **Aceptar**. Se agrega un nuevo archivo que contiene la clase ReadOnlyFile.

5. En la parte superior del archivo ReadOnlyFile.cs o ReadOnlyFile.vb, agregue el código siguiente para importar los espacios de nombres System.IO y System.Dynamic.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

6. El objeto dinámico personalizado usa una enumeración para determinar los criterios de búsqueda. Antes de la instrucción de clase, agregue la siguiente definición de enumeración.

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

7. Actualice la instrucción de clase para heredar la clase `DynamicObject`, como se muestra en el ejemplo de código siguiente.

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

8. Agregue el código siguiente a la clase `ReadOnlyFile` para definir un campo privado para la ruta de acceso y un constructor para la clase `ReadOnlyFile`.

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

```
' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub
```

9. Agregue el método `GetPropertyValue` siguiente a la clase `ReadOnlyFile`. El método `GetPropertyValue` toma como entrada criterios de búsqueda y devuelve las líneas de un archivo de texto que coinciden con los criterios de búsqueda. Los métodos dinámicos proporcionados por la clase `ReadOnlyFile` llaman al método `GetPropertyValue` para recuperar los resultados correspondientes.

```
public List<string> GetPropertyValue(string propertyName,
                                     StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                     bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

```
Public Function GetPropertyValue(ByVal propertyName As String,
                                 Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                                 Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCase(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCase(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function
```

10. Después del método `GetPropertyValue`, agregue el código siguiente para invalidar el método TryGetMember de la clase DynamicObject. Se llama al método TryGetMember cuando se solicita un miembro de una clase dinámica y no se especifican argumentos. El argumento `binder` contiene información sobre el miembro al que se hace referencia y el argumento `result` hace referencia al resultado devuelto para el miembro especificado. El método TryGetMember devuelve un valor booleano que devuelve `true` si el miembro solicitado existe. En caso contrario, devuelve `false`.

```
// Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

```
' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function
```

11. Después del método `TryGetMember`, agregue el código siguiente para invalidar el método TryInvokeMember de la clase DynamicObject. Se llama al método TryInvokeMember cuando se solicita un miembro de una clase dinámica con argumentos. El argumento `binder` contiene información sobre el miembro al que se hace referencia y el argumento `result` hace referencia al resultado devuelto para el miembro especificado. El argumento `args` contiene una matriz de los argumentos que se pasan al miembro. El método TryInvokeMember devuelve un valor booleano que devuelve `true` si el miembro solicitado existe. En caso contrario, devuelve `false`.

    La versión personalizada del método `TryInvokeMember` espera que el primer argumento sea un valor del enumerador `StringSearchOption` que se ha definido en un paso anterior. El método `TryInvokeMember` espera que el segundo argumento sea un valor booleano. Si uno o ambos argumentos son valores válidos, se pasan al método `GetPropertyValue` para recuperar los resultados.

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```
' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                          ByVal args() As Object,
                                          ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function
```

12. Guarde y cierre el archivo.

Para crear un archivo de texto de ejemplo

1. Haga clic con el botón derecho en el proyecto DynamicSample, seleccione **Agregar** y, después, haga clic en **Nuevo elemento**. En el panel **Plantillas instaladas** seleccione **General** y, luego, la plantilla **Archivo de texto**. Deje el nombre predeterminado TextFile1.txt en el cuadro **Nombre** y haga clic en **Agregar**. Se agregará un archivo de texto nuevo al proyecto.

2. Copie el texto siguiente en el archivo TextFile1.txt.

```
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

3. Guarde y cierre el archivo.

Para crear una aplicación de ejemplo que usa el objeto dinámico personalizado

1. En el **Explorador de soluciones**, haga doble clic en el archivo Module1.vb si usa Visual Basic o en el archivo Program.cs si usa Visual C#.

2. Agregue el código siguiente al procedimiento Main para crear una instancia de la clase `ReadOnlyFile` para el archivo TextFile1.txt. El código usa el enlace en tiempo de ejecución para llamar a miembros dinámicos y recuperar líneas de texto que contienen la cadena "Customer".

```
dynamic rFile = new ReadOnlyFile(@"..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

```
Dim rFile As Object = New ReadOnlyFile("..\..\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("----------------------------")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next
```

3. Guarde el archivo y presione CTRL+F5 para compilar y ejecutar la aplicación.

Llamar a una biblioteca de lenguaje dinámico

El siguiente proyecto que cree en este tutorial tendrá acceso a una biblioteca escrita en el lenguaje dinámico IronPython.

Para crear una clase dinámica personalizada

1. En el menú **Archivo** de Visual Studio, apunte a **Nuevo** y haga clic en **Proyecto**.

2. En el panel **Tipos de proyecto** del cuadro de diálogo **Nuevo proyecto**, asegúrese de que esté seleccionado **Windows**. Seleccione **Aplicación de consola** en el panel **Plantillas**. En el cuadro **Nombre**, escriba `DynamicIronPythonSample` y haga clic en **Aceptar**. Se crea el proyecto.

3. Si usa Visual Basic, haga clic con el botón derecho en el proyecto DynamicIronPythonSample y luego haga clic en **Propiedades**. Haga clic en la pestaña **Referencias**. Haga clic en el botón **Agregar**. Si usa Visual C#, en el **Explorador de soluciones**, haga clic con el botón derecho en la carpeta **Referencias** y, después, haga clic en **Agregar referencia**.

4. En la pestaña **Examinar**, vaya a la carpeta donde están instaladas las bibliotecas de IronPython. Por ejemplo, C:\Program Files\IronPython 2.6 para .NET 4.0. Seleccione las bibliotecas **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Scripting.dll** y **Microsoft.Dynamic.dll**. Haga clic en **Aceptar**.

5. Si usa Visual Basic, edite el archivo Module1.vb. Si usa Visual C#, edite el archivo Program.cs.

6. En la parte superior del archivo, agregue el código siguiente para importar los espacios de nombres `Microsoft.Scripting.Hosting` y `IronPython.Hosting` de las bibliotecas de IronPython.

```
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

```
Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
```

7. En el método Main, agregue el código siguiente para crear un objeto `Microsoft.Scripting.Hosting.ScriptRuntime` que hospede las bibliotecas de IronPython. El objeto

`ScriptRuntime` carga el módulo de biblioteca de IronPython random.py.

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.6 for .NET 4.0\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

```
' Set the current directory to the IronPython libraries.
My.Computer.FileSystem.CurrentDirectory =
   My.Computer.FileSystem.SpecialDirectories.ProgramFiles &
   "\IronPython 2.6 for .NET 4.0\Lib"

' Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")
```

8. Una vez que el código haya cargado el módulo random.py, agregue el código siguiente para crear una matriz de enteros. La matriz se pasa al método `shuffle` del módulo random.py, que ordena aleatoriamente los valores de la matriz.

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("-------------------");
}
```

```
' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("-------------------")
Next
```

9. Guarde el archivo y presione CTRL+F5 para compilar y ejecutar la aplicación.

Vea también

- System.Dynamic

- System.Dynamic.DynamicObject
- Uso de tipo dinámico
- Enlace en tiempo de compilación y en tiempo de ejecución
- dynamic
- Implementar interfaces dinámicas (PDF descargable de Microsoft TechNet)

Clases y estructuras (guía de programación de C#)

29/06/2020 • 12 minutes to read • Edit Online

Las clases (class) y estructuras (struct) son dos de las construcciones básicas de Common Type System en .NET. Cada una de ellas es básicamente una estructura de datos que encapsula un conjunto de datos y comportamientos que forman un conjunto como una unidad lógica. Los datos y comportamientos son los *miembros* de la clase o estructura, e incluyen sus métodos, propiedades y eventos, entre otros elementos, como se muestra más adelante en este tema.

Una declaración de clase o estructura es como un plano que se utiliza para crear instancias u objetos en tiempo de ejecución. Si define una clase o estructura llamada `Person`, `Person` es el nombre del tipo. Si declara e inicializa una variable `p` de tipo `Person`, se dice que `p` es un objeto o instancia de `Person`. Se pueden crear varias instancias del mismo tipo `Person`, y cada instancia tiene diferentes valores en sus propiedades y campos.

Una clase es un tipo de referencia. Cuando se crea un objeto de la clase, la variable a la que se asigna el objeto contiene solo una referencia a esa memoria. Cuando la referencia de objeto se asigna a una nueva variable, la nueva variable hace referencia al objeto original. Los cambios realizados en una variable se reflejan en la otra variable porque ambas hacen referencia a los mismos datos.

Una estructura es un tipo de valor. Cuando se crea una estructura, la variable a la que se asigna la estructura contiene los datos reales de ella. Cuando la estructura se asigna a una nueva variable, se copia. Por lo tanto, la nueva variable y la variable original contienen dos copias independientes de los mismos datos. Los cambios realizados en una copia no afectan a la otra copia.

En general, las clases se utilizan para modelar comportamientos más complejos, o datos que se prevén modificar después de haber creado un objeto de clase. Las estructuras son más adecuadas para las estructuras de datos pequeñas que contienen principalmente datos que no se prevén modificar después de haber creado la estructura.

Para más información, vea Clases, Objectos y Tipos de estructura.

Ejemplo

En el ejemplo siguiente, `CustomClass` en el espacio de nombres `ProgrammingGuide` tiene tres miembros: un constructor de instancia, una propiedad denominada `Number` y un método denominado `Multiply`. El método `Main` de la clase `Program` crea una instancia (objeto) de `CustomClass`, y se puede acceder a la propiedad y al método del objeto mediante una notación de puntos.

```
using System;

namespace ProgrammingGuide
{
   // Class definition.
   public class CustomClass
   {
      // Class members.
      //
      // Property.
      public int Number { get; set; }

      // Method.
      public int Multiply(int num)
      {
          return num * Number;
      }

      // Instance Constructor.
      public CustomClass()
      {
          Number = 0;
      }
   }

   // Another class definition that contains Main, the program entry point.
   class Program
   {
      static void Main(string[] args)
      {
         // Create an object of type CustomClass.
         CustomClass custClass = new CustomClass();

         // Set the value of the public property.
         custClass.Number = 27;

         // Call the public method.
         int result = custClass.Multiply(4);
         Console.WriteLine($"The result is {result}.");
      }
   }
}
// The example displays the following output:
//      The result is 108.
```

Encapsulación

A veces se hace referencia a la *encapsulación* como el primer pilar o principio de la programación orientada a objetos. Según el principio de encapsulación, una clase o una estructura pueden especificar hasta qué punto se puede acceder a sus miembros para codificar fuera de la clase o la estructura. No se prevé el uso de los métodos y las variables fuera de la clase, o el ensamblado puede ocultarse para limitar el potencial de los errores de codificación o de los ataques malintencionados.

Para más información sobre las clases, vea Clases y Objetos.

Miembros

Todos los métodos, campos, constantes, propiedades y eventos deben declararse dentro de un tipo; se les denomina *miembros* del tipo. En C#, no hay métodos ni variables globales como en otros lenguajes. Incluso un punto de entrada del programa, el método `Main`, debe declararse dentro de una clase o estructura. La lista siguiente incluye los diversos tipos de miembros que se pueden declarar en una clase o estructura.

- Campos

- Constantes
- Propiedades
- Métodos
- Constructores
- Eventos
- Finalizadores
- Indizadores
- Operadores
- Tipos anidados

Accesibilidad

Algunos métodos y propiedades están diseñados para ser invocables y accesibles desde el código fuera de la clase o la estructura, lo que se conoce como *código de cliente*. Otros métodos y propiedades pueden estar indicados exclusivamente para utilizarse en la propia clase o estructura. Es importante limitar la accesibilidad del código, a fin de que solo el código de cliente previsto pueda acceder a él. Puede usar los modificadores de acceso public, protected, internal, protected internal, private y private protected para especificar hasta qué punto los tipos y sus miembros son accesibles para el código de cliente. La accesibilidad predeterminada es `private`. Para obtener más información, consulte Modificadores de acceso.

Herencia

Las clases (pero no las estructuras) admiten el concepto de herencia. Una clase que deriva de otra clase (la *clase base*) contiene automáticamente todos los miembros públicos, protegidos e internos de la clase base, salvo sus constructores y finalizadores. Para más información, vea Herencia y Polimorfismo.

Las clases pueden declararse como abstract, lo que significa que uno o varios de sus métodos no tienen ninguna implementación. Aunque no se pueden crear instancias de clases abstractas directamente, pueden servir como clases base para otras clases que proporcionan la implementación que falta. Las clases también pueden declararse como sealed para evitar que otras clases hereden de ellas. Para más información, vea Clases y miembros de clase abstractos y sellados.

Interfaces

Las clases y las estructuras pueden heredar varias interfaces. Heredar de una interfaz significa que el tipo implementa todos los métodos definidos en la interfaz. Para más información, vea Interfaces.

Tipos genéricos

Las clases y estructuras pueden definirse con uno o varios parámetros de tipo. El código de cliente proporciona el tipo cuando crea una instancia del tipo. Por ejemplo, la clase List<T> del espacio de nombres System.Collections.Generic se define con un parámetro de tipo. El código de cliente crea una instancia de `List<string>` o `List<int>` para especificar el tipo que contendrá la lista. Para más información, vea Genéricos.

Tipos estáticos

Las clases (pero no las estructuras) pueden declararse como static. Una clase estática puede contener solo miembros estáticos y no se puede crear una instancia de ellos con la palabra clave new. Una copia de la clase se carga en memoria cuando se carga el programa, y sus miembros son accesibles a través del nombre de clase. Las clases y estructuras pueden contener miembros estáticos. Para más información, vea Clases estáticas y sus miembros.

Tipos anidados

Una clase o estructura se puede anidar dentro de otra clase o estructura. Para obtener más información, consulte Tipos anidados.

Tipos parciales

Puede definir parte de una clase, estructura o método en un archivo de código y otra parte en un archivo de código independiente. Para más información, vea Clases y métodos parciales.

Inicializadores de objeto

Puede crear instancias de objetos de clase o estructura y de colecciones de objetos e iniciarlizarlos, sin llamar de forma explícita a su constructor. Para más información, vea Inicializadores de objeto y de colección.

Tipos anónimos

En situaciones donde no es conveniente o necesario crear una clase con nombre, por ejemplo al rellenar una lista con estructuras de datos que no tiene que conservar o pasar a otro método, utilice los tipos anónimos. Para más información, vea Tipos anónimos.

Métodos de extensión.

Puede "extender" una clase sin crear una clase derivada mediante la creación de un tipo independiente cuyos métodos pueden llamarse como si pertenecieran al tipo original. Para más información, vea Métodos de extensión.

Variables locales con asignación implícita de tipos

Dentro de un método de clase o estructura, puede utilizar tipos implícitos para indicar al compilador que determine el tipo correcto en tiempo de compilación. Para más información, vea Variables locales con asignación implícita de tipos.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#

Clases (Guía de programación de C#)

29/06/2020 • 9 minutes to read • Edit Online

Tipos de referencia

Un tipo que se define como una clase, es un *tipo de referencia*. Al declarar una variable de un tipo de referencia en tiempo de ejecución, esta contendrá el valor null hasta que se cree expresamente una instancia de la clase mediante el operador new o se le asigne un objeto de un tipo compatible que se ha creado en otro lugar, tal y como se muestra en el ejemplo siguiente:

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.
MyClass mc2 = mc;
```

Cuando se crea el objeto, se asigna suficiente memoria en el montón administrado para ese objeto específico y la variable solo contiene una referencia a la ubicación de dicho objeto. Los tipos del montón administrado producen sobrecarga cuando se asignan y cuando los reclama la función de administración de memoria automática de CLR, conocida como *recolección de elementos no utilizados*. En cambio, la recolección de elementos no utilizados también está muy optimizada y no crea problemas de rendimiento en la mayoría de los escenarios. Para obtener más información sobre la recolección de elementos no utilizados, vea Administración automática de la memoria y recolección de elementos no utilizados.

Declarar clases

Las clases se declaran mediante la palabra clave class seguida por un identificador único, como se muestra en el siguiente ejemplo:

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

La palabra clave `class` va precedida del nivel de acceso. Como en este caso se usa public, cualquier usuario puede crear instancias de esta clase. El nombre de la clase sigue a la palabra clave `class`. El nombre de la clase debe ser un nombre de identificador de C# válido. El resto de la definición es el cuerpo de la clase, donde se definen los datos y el comportamiento. Los campos, las propiedades, los métodos y los eventos de una clase se denominan de manera colectiva *miembros de clase*.

Creación de objetos

Aunque a veces se usan indistintamente, una clase y un objeto son cosas diferentes. Una clase define un tipo de objeto, pero no es un objeto en sí. Un objeto es una entidad concreta basada en una clase y, a veces, se conoce como una instancia de una clase.

Los objetos se pueden crear usando la palabra clave new, seguida del nombre de la clase en la que se basará el objeto, como en este ejemplo:

```
Customer object1 = new Customer();
```

Cuando se crea una instancia de una clase, se vuelve a pasar al programador una referencia al objeto. En el ejemplo anterior, `object1` es una referencia a un objeto que se basa en `Customer`. Esta referencia apunta al objeto nuevo, pero no contiene los datos del objeto. De hecho, puede crear una referencia de objeto sin tener que crear ningún objeto:

```
Customer object2;
```

No se recomienda crear referencias de objeto como esta, que no hace referencia a ningún objeto, ya que, si se intenta obtener acceso a un objeto a través de este tipo de referencia, se producirá un error en tiempo de ejecución. Pero dicha referencia puede haberse creado para hacer referencia a un objeto, ya sea creando un nuevo objeto o asignándola a un objeto existente, como en el siguiente ejemplo:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

Este código crea dos referencias de objeto que hacen referencia al mismo objeto. Por lo tanto, los cambios efectuados en el objeto mediante `object3` se reflejan en los usos posteriores de `object4`. Dado que los objetos basados en clases se tratan por referencia, las clases se denominan "tipos de referencia".

Herencia de clases

Las clases admiten completamente la *herencia*, una característica fundamental de la programación orientada a objetos. Al crear una clase, puede heredar de cualquier otra interfaz o clase que no esté definida como sealed; y otras clases pueden heredar de su clase e invalidar los métodos virtuales de la clase.

La herencia se consigue mediante una *derivación*, en la que se declara una clase mediante una *clase base*, desde la que hereda los datos y el comportamiento. Una clase base se especifica anexando dos puntos y el nombre de la clase base seguido del nombre de la clase derivada, como en el siguiente ejemplo:

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

Cuando una clase declara una clase base, hereda todos los miembros de la clase base excepto los constructores. Para obtener más información, vea Herencia.

A diferencia de C++, una clase de C# solo puede heredar directamente de una clase base. En cambio, dado que una clase base puede heredar de otra clase, una clase podría heredar indirectamente varias clases base. Además, una clase puede implementar directamente más de una interfaz. Para obtener más información, vea Interfaces.

Una clase puede declararse abstracta. Una clase abstracta contiene métodos abstractos que tienen una definición de firma, pero no tienen ninguna implementación. No se pueden crear instancias de las clases abstractas. Solo se pueden usar a través de las clases derivadas que implementan los métodos abstractos. Por el contrario, la clase sealed no permite que otras clases se deriven de ella. Para más información, vea Clases y miembros de clase abstractos y sellados.

Las definiciones de clase se pueden dividir entre distintos archivos de código fuente. Para más información, vea Clases y métodos parciales.

Ejemplo

En el ejemplo siguiente se define una clase pública que contiene una propiedad implementada automáticamente, un método y un método especial denominado constructor. Para obtener más información, consulta los temas Propiedades, Métodos y Constructores. Luego, se crea una instancia de las instancias de la clase con la palabra clave `new`.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#

- Programación orientada a objetos
- Polimorfismo
- Nombres de identificador
- Miembros
- Métodos
- Constructores
- Finalizadores
- Objects

Objetos (Guía de programación de C#)

29/06/2020 • 9 minutes to read • Edit Online

Una definición de clase o estructura es como un plano que especifica qué puede hacer el tipo. Un objeto es básicamente un bloque de memoria que se ha asignado y configurado de acuerdo con el plano. Un programa puede crear muchos objetos de la misma clase. Los objetos también se denominan instancias y pueden almacenarse en una variable con nombre, o en una matriz o colección. El código de cliente es el código que usa estas variables para llamar a los métodos y acceder a las propiedades públicas del objeto. En un lenguaje orientado a objetos, como C#, un programa típico consta de varios objetos que interactúan dinámicamente.

> **NOTE**
>
> Los tipos estáticos se comportan de forma diferente a lo que se describe aquí. Para más información, vea Clases estáticas y sus miembros.

Instancias de estructura frente a Instancias de clase

Puesto que las clases son tipos de referencia, una variable de un objeto de clase contiene una referencia a la dirección del objeto del montón administrado. Si se asigna un segundo objeto del mismo tipo al primer objeto, ambas variables hacen referencia al objeto de esa dirección. Este punto se analiza con más detalle más adelante en este tema.

Las instancias de clases se crean mediante el operador new. En el ejemplo siguiente, `Person` es el tipo, y `person1` y `person 2` son instancias u objetos de ese tipo.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
    person1 Name = Molly Age = 16
*/
```

Dado que las estructuras son tipos de valor, una variable de un objeto de estructura contiene una copia de todo el objeto. También se pueden crear instancias de estructuras usando el operador `new`, pero esto no resulta necesario, como se muestra en el ejemplo siguiente:

```
public struct Person
{
    public string Name;
    public int Age;
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public class Application
{
    static void Main()
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create  new struct object. Note that  struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is  copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
  Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
*/
```

La memoria para `p1` y `p2` se asigna en la pila de subprocesos. Esta memoria se reclama junto con el tipo o método en el que se declara. Este es uno de los motivos por los que se copian las estructuras en la asignación. Por el contrario, la memoria que se asigna a una instancia de clase la reclama automáticamente (recolección de elementos no utilizados) Common Language Runtime cuando todas las referencias al objeto se han salido del ámbito. No es posible destruir de forma determinante un objeto de clase como en C++. Para obtener más información sobre la recolección de elementos no utilizados en .NET, vea Recolección de elementos no utilizados.

> **NOTE**
> La asignación y desasignación de memoria en el montón administrado están muy optimizadas en Common Language Runtime. En la mayoría de los casos, no existe ninguna diferencia significativa en el costo de rendimiento entre asignar una instancia de clase en el montón y asignar una instancia de estructura en la pila.

Identidad de objeto frente a igualdad de valores

Cuando se comparan dos objetos para comprobar si son iguales, primero debe determinar si quiere saber si las dos variables representan el mismo objeto en la memoria o si los valores de uno o varios de sus campos son

equivalentes. Si tiene previsto comparar valores, debe tener en cuenta si los objetos son instancias de tipos de valor (estructuras) o tipos de referencia (clases, delegados y matrices).

- Para determinar si dos instancias de clase hacen referencia a la misma ubicación en la memoria (lo que significa que tienen la misma *identidad*), use el método estático Equals. (System.Object es la clase base implícita para todos los tipos de valor y tipos de referencia, incluidas las clases y estructuras definidas por el usuario).
- Para determinar si los campos de instancia de dos instancias de estructura presentan los mismos valores, use el método ValueType.Equals. Dado que todas las estructuras heredan implícitamente de System.ValueType, se llama al método directamente en el objeto, como se muestra en el ejemplo siguiente:

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

La implementación System.ValueType de `Equals` usa la reflexión porque debe poder determinar cuáles son los campos en cualquier estructura. Al crear sus propias estructuras, invalide el método `Equals` para proporcionar un algoritmo de igualdad eficaz específico de su tipo.

- Para determinar si los valores de los campos de dos instancias de clase son iguales, puede usar el método Equals o el operador ==. En cambio, úselos solo si la clase los ha invalidado o sobrecargado para proporcionar una definición personalizada de lo que significa "igualdad" para los objetos de ese tipo. La clase también puede implementar la interfaz IEquatable<T> o la interfaz IEqualityComparer<T>. Ambas interfaces proporcionan métodos que pueden servir para comprobar la igualdad de valores. Al diseñar sus propias clases que invaliden `Equals`, asegúrese de seguir las instrucciones descritas en Procedimiento: Definición de la igualdad de valores para un tipo y Object.Equals(Object).

Secciones relacionadas

Para obtener más información:

- Clases
- Constructores
- Finalizadores
- Eventos

Vea también

- Guía de programación de C#
- object
- Herencia
- class
- Tipos de estructura
- new (operador)
- Sistema de tipos comunes

Herencia (Guía de programación de C#)

18/03/2020 • 11 minutes to read • Edit Online

La herencia, junto con la encapsulación y el polimorfismo, es una de las tres características principales de la programación orientada a objetos. La herencia permite crear clases que reutilizan, extienden y modifican el comportamiento definido en otras clases. La clase cuyos miembros se heredan se denomina *clase base* y la clase que hereda esos miembros se denomina *clase derivada*. Una clase derivada solo puede tener una clase base directa, pero la herencia es transitiva. Si `ClassC` se deriva de `ClassB` y `ClassB` se deriva de `ClassA`, `ClassC` hereda los miembros declarados en `ClassB` y `ClassA`.

> **NOTE**
>
> Los structs no admiten la herencia, pero pueden implementar interfaces. Para más información, vea Interfaces.

Conceptualmente, una clase derivada es una especialización de la clase base. Por ejemplo, si tiene una clase base `Animal`, podría tener una clase derivada denominada `Mammal` y otra clase derivada denominada `Reptile`. `Mammal` es `Animal` y `Reptile` también es `Animal`, pero cada clase derivada representa especializaciones diferentes de la clase base.

Las declaraciones de interfaz pueden definir una implementación predeterminada de sus miembros. Estas implementaciones las heredan las interfaces derivadas y las clases que implementan esas interfaces. Para obtener más información sobre los métodos de interfaz predeterminados, vea el artículo sobre interfaces en la sección de referencia del lenguaje.

Cuando se define una clase para que derive de otra clase, la clase derivada obtiene implícitamente todos los miembros de la clase base, salvo sus constructores y sus finalizadores. La clase derivada reutiliza el código de la clase base sin tener que volver a implementarlo. Puede agregar más miembros en la clase derivada. La clase derivada amplía la funcionalidad de la clase base.

En la ilustración siguiente se muestra una clase `WorkItem` que representa un elemento de trabajo de un proceso empresarial. Como con todas las clases, se deriva de System.Object y hereda todos sus métodos. `WorkItem` agrega cinco miembros propios. Estos miembros incluyen un constructor, porque los constructores no se heredan. La clase `ChangeRequest` hereda de `WorkItem` y representa un tipo concreto de elemento de trabajo. `ChangeRequest` agrega dos miembros más a los miembros que hereda de `WorkItem` y de Object. Debe agregar su propio constructor y además agrega `originalItemID`. La propiedad `originalItemID` permite que la instancia `ChangeRequest` se asocie con el `WorkItem` original al que se aplica la solicitud de cambio.

En el ejemplo siguiente se muestra cómo se expresan en C# las relaciones de clase de la ilustración anterior. En el ejemplo también se muestra cómo `WorkItem` reemplaza el método virtual Object.ToString y cómo la clase `ChangeRequest` hereda la implementación `WorkItem` del método. En el primer bloque se definen las clases:

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
```

```
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemId is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}
```

En el bloque siguiente se muestra cómo usar las clases base y derivadas:

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
    new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

Métodos abstractos y virtuales

Cuando una clase base declara un método como `virtual`, una clase derivada puede aplicar `override` al método con una implementación propia. Si una clase base declara un miembro como `abstract`, ese método se debe reemplazar en todas las clases no abstractas que hereden directamente de dicha clase. Si una clase derivada es abstracta, hereda los miembros abstractos sin implementarlos. Los miembros abstractos y virtuales son la base del polimorfismo, que es la segunda característica principal de la programación orientada a objetos. Para obtener más información, vea Polimorfismo .

Clases base abstractas

Puede declarar una clase como abstracta si quiere impedir la creación directa de instancias mediante el operador new. Una clase abstracta solo se puede usar si a partir de ella se deriva una clase nueva. Una clase abstracta puede contener una o más firmas de método que, a su vez, se declaran como abstractas. Estas firmas especifican los parámetros y el valor devuelto, pero no tienen ninguna implementación (cuerpo del método). Una clase abstracta no tiene que contener miembros abstractos, pero si lo hace, la clase se debe declarar como abstracta. Las clases derivadas que no son abstractas deben proporcionar la implementación para todos los métodos abstractos de una clase base abstracta. Para más información, vea Clases y miembros de clase abstractos y sellados.

Interfaces

Una *interfaz* es un tipo de referencia que define un conjunto de miembros. Todas las clases y estructuras que implementan esa interfaz deben implementar ese conjunto de miembros. Una interfaz puede definir una implementación predeterminada para todos o ninguno de estos miembros. Una clase puede implementar varias interfaces, aunque solo puede derivar de una única clase base directa.

Las interfaces se usan para definir funciones específicas para clases que no tienen necesariamente una relación "es un/una". Por ejemplo, la interfaz System.IEquatable<T> se puede implementar mediante cualquier clase o estructura para determinar si dos objetos del tipo son equivalentes (pero el tipo define la

equivalencia). IEquatable<T> no implica el mismo tipo de relación "es un/una" que existe entre una clase base y una clase derivada (por ejemplo, `Mammal` es `Animal`). Para más información, vea Interfaces.

Impedir la derivación adicional

Una clase puede impedir que otras clases hereden de ella, o de cualquiera de sus miembros, si se declara a sí misma o al miembro como `sealed`. Para más información, vea Clases y miembros de clase abstractos y sellados.

Clase derivada que oculta miembros de clase base

Una clase derivada puede ocultar miembros de clase base si declara los miembros con el mismo nombre y firma. Se puede usar el modificador `new` para indicar de forma explícita que el miembro no está diseñado para reemplazar al miembro base. No es necesario usar `new`, pero se generará una advertencia del compilador si no se usa `new`. Para obtener más información, vea Control de versiones con las palabras clave Override y New y Saber cuándo usar las palabras clave Override y New.

Vea también

- Guía de programación de C#
- Clases y structs
- class

Polimorfismo (Guía de programación de C#)

29/06/2020 • 14 minutes to read • Edit Online

El polimorfismo suele considerarse el tercer pilar de la programación orientada a objetos, después de la encapsulación y la herencia. Polimorfismo es una palabra griega que significa "con muchas formas" y tiene dos aspectos diferentes:

- En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como objetos de una clase base en lugares como parámetros de métodos y colecciones o matrices. Cuando se produce este polimorfismo, el tipo declarado del objeto ya no es idéntico a su tipo en tiempo de ejecución.
- Las clases base pueden definir e implementar *métodos* virtuales, y las clases derivadas pueden invalidarlos, lo que significa que pueden proporcionar su propia definición e implementación. En tiempo de ejecución, cuando el código de cliente llama al método, CLR busca el tipo en tiempo de ejecución del objeto e invoca esa invalidación del método virtual. En el código fuente puede llamar a un método en una clase base y hacer que se ejecute una versión del método de la clase derivada.

Los métodos virtuales permiten trabajar con grupos de objetos relacionados de manera uniforme. Por ejemplo, supongamos que tiene una aplicación de dibujo que permite a un usuario crear varios tipos de formas en una superficie de dibujo. En tiempo de compilación, no sabe qué tipos específicos de formas creará el usuario. Sin embargo, la aplicación tiene que realizar el seguimiento de los distintos tipos de formas que se crean, y tiene que actualizarlos en respuesta a las acciones del mouse del usuario. Para solucionar este problema en dos pasos básicos, puede usar el polimorfismo:

1. Crear una jerarquía de clases en la que cada clase de forma específica deriva de una clase base común.
2. Usar un método virtual para invocar el método apropiado en una clase derivada mediante una sola llamada al método de la clase base.

Primero, cree una clase base llamada `Shape` y clases derivadas como `Rectangle`, `Circle` y `Triangle`. Dé a la clase `Shape` un método virtual llamado `Draw` e invalídelo en cada clase derivada para dibujar la forma determinada que la clase representa. Cree un objeto `List<Shape>` y agréguele una instancia de `Circle`, `Triangle` y `Rectangle`.

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

Para actualizar la superficie de dibujo, use un bucle foreach para iterar por la lista y llamar al método `Draw` en cada objeto `Shape` de la lista. Aunque cada objeto de la lista tenga un tipo declarado de `Shape`, se invocará el tipo en tiempo de ejecución (la versión invalidada del método en cada clase derivada).

```
// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used whereever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
*/
```

En C#, cada tipo es polimórfico porque todos los tipos, incluidos los definidos por el usuario, heredan de Object.

Introducción al polimorfismo

Miembros virtuales

Cuando una clase derivada hereda de una clase base, obtiene todos los métodos, campos, propiedades y eventos de la clase base. El diseñador de la clase derivada puede tener diferentes opciones para el comportamiento de los métodos virtuales:

- La clase derivada puede invalidar los miembros virtuales de la clase base, y definir un comportamiento nuevo.
- La clase derivada hereda el método de clase base más cercano sin invalidarlo, para conservar el comportamiento existente, pero permite que más clases derivadas invaliden el método.
- La clase derivada puede definir una nueva implementación no virtual de esos miembros que oculte las implementaciones de la clase base.

Una clase derivada puede invalidar un miembro de la clase base si este se declara como virtual o abstracto. El miembro derivado debe usar la palabra clave override para indicar explícitamente que el propósito del método es participar en una invocación virtual. El siguiente fragmento de código muestra un ejemplo:

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Los campos no pueden ser virtuales; solo pueden serlo los métodos, propiedades, eventos e indizadores. Cuando una clase derivada invalida un miembro virtual, se llama a ese miembro aunque se acceda a una instancia de esa clase como una instancia de la clase base. El siguiente fragmento de código muestra un ejemplo:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Los métodos y propiedades virtuales permiten a las clases derivadas extender una clase base sin necesidad de usar la implementación de clase base de un método. Para obtener más información, consulte Control de versiones con las palabras clave Override y New. Una interfaz proporciona otra manera de definir un método o conjunto de métodos cuya implementación se deja a las clases derivadas. Para más información, vea Interfaces.

Ocultación de miembros de clase base con miembros nuevos

Si quiere que la clase derivada tenga un miembro con el mismo nombre que el de un miembro de una clase base, puede usar la palabra clave new para ocultar el miembro de clase base. La palabra clave `new` se coloca antes que el tipo devuelto del miembro de la clase que se está reemplazando. El siguiente fragmento de código muestra un ejemplo:

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Se puede acceder a los miembros de la clase base ocultos desde el código de cliente si se convierte la instancia de la clase derivada en una instancia de la clase base. Por ejemplo:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

Evasión de que las clases derivadas invaliden los miembros virtuales

Los miembros virtuales siguen siendo virtuales con independencia de cuántas clases se hayan declarado entre el miembro virtual y la clase que originalmente lo haya declarado. Si la clase `A` declara un miembro virtual y la clase `B` deriva de `A`, y la clase `C` de `B`, la clase `C` hereda el miembro virtual y puede invalidarlo, independientemente de que la clase `B` haya declarado una invalidación para ese miembro. El siguiente fragmento de código muestra un ejemplo:

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

Una clase derivada puede detener la herencia virtual al declarar una invalidación como sealed. Para detener la herencia, es necesario colocar la palabra clave `sealed` antes de la palabra clave `override` en la declaración del miembro de la clase. El siguiente fragmento de código muestra un ejemplo:

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

En el ejemplo anterior, el método `DoWork` ya no es virtual para ninguna clase que se derive de `C`. Sigue siendo virtual para las instancias de `C`, aunque se conviertan al tipo `B` o al tipo `A`. Los métodos sellados se pueden reemplazar por clases derivadas mediante la palabra clave `new`, como se muestra en el ejemplo siguiente:

```
public class D : C
{
    public new void DoWork() { }
}
```

En este caso, si se llama a `DoWork` en `D` con una variable de tipo `D`, se llama a la nueva instancia de `DoWork`. Si se usa una variable de tipo `C`, `B` o `A` para acceder a una instancia de `D`, la llamada a `DoWork` seguirá las reglas de herencia virtual y enrutará esas llamadas a la implementación de `DoWork` en la clase `C`.

Acceso a miembros virtuales de clases base desde clases derivadas

Una clase derivada que ha reemplazado o invalidado un método o propiedad puede seguir accediendo al método o propiedad en la clase base usando la siguiente palabra clave `base`. El siguiente fragmento de código muestra un ejemplo:

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

Para obtener más información, vea base.

> **NOTE**
>
> Se recomienda que las máquinas virtuales usen `base` para llamar a la implementación de la clase base de ese miembro en su propia implementación. Dejar que se produzca el comportamiento de la clase base permite a la clase derivada concentrarse en implementar el comportamiento específico de la clase derivada. Si no se llama a la implementación de la clase base, depende de la clase derivada hacer que su comportamiento sea compatible con el de la clase base.

En esta sección

- Control de versiones con las palabras clave Override y New
- Saber cuándo utilizar las palabras clave Override y New
- Invalidación del método ToString

Vea también

- Guía de programación de C#
- Herencia
- Clases y miembros de clase abstractos y sellados
- Métodos
- Eventos
- Propiedades
- Indizadores
- Tipos

Control de versiones con las palabras clave Override y New (Guía de programación de C#)

29/06/2020 • 10 minutes to read • Edit Online

El lenguaje C# está diseñado para que las versiones entre clases base y derivadas de diferentes bibliotecas puedan evolucionar y mantener la compatibilidad con versiones anteriores. Esto significa, por ejemplo, que la introducción de un nuevo miembro en una clase base con el mismo nombre que un miembro de una clase derivada es totalmente compatible con C# y no lleva a un comportamiento inesperado. Además, implica que una clase debe declarar explícitamente si un método está pensado para reemplazar un método heredado o si se trata de un nuevo método que oculta un método heredado de nombre similar.

En C#, las clases derivadas pueden contener métodos con el mismo nombre que los métodos de clase base.

- Si el método de la clase derivada no va precedido por las palabras clave new u override, el compilador emite una advertencia y el método se comporta como si la palabra clave `new` estuviese presente.
- Si el método de la clase derivada va precedido de la palabra clave `new`, el método se define como independiente del método de la clase base.
- Si el método de la clase derivada va precedido de la palabra clave `override`, los objetos de la clase derivada llamarán a ese método y no al método de la clase base.
- Para aplicar la palabra clave `override` al método de la clase derivada, se debe definir el método de clase base como virtual.
- El método de clase base puede llamarse desde dentro de la clase derivada mediante la palabra clave `base` .
- Las palabras clave `override`, `virtual` y `new` también pueden aplicarse a propiedades, indexadores y eventos.

De forma predeterminada, los métodos de C# no son virtuales. Si se declara un método como virtual, toda clase que hereda el método puede implementar su propia versión Para que un método sea virtual, se usa el modificador `virtual` en la declaración del método de la clase base. La clase derivada puede luego reemplazar el método base virtual mediante la palabra clave `override` u ocultar el método virtual en la clase base mediante la palabra clave `new`. Si no se especifican las palabras clave `override` o `new`, el compilador emite una advertencia y el método de la clase derivada oculta el método de la clase base.

Para demostrar esto en la práctica, supongamos por un momento que la compañía A ha creado una clase denominada `GraphicsClass`, que su programa usa. La siguiente es `GraphicsClass`:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Su compañía usa esta clase y usted la usa para derivar su propia clase, agregando un nuevo método:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

La aplicación se usa sin problemas, hasta que la compañía A lanza una nueva versión de `GraphicsClass`, que es similar al código siguiente:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

La nueva versión de `GraphicsClass` contiene ahora un método denominado `DrawRectangle`. Inicialmente, no sucede nada. La nueva versión sigue siendo compatible a nivel binario con la versión anterior. Cualquier software que haya implementado seguirá funcionando, aunque la nueva clase se instale en esos sistemas informáticos. Cualquier llamada existente al método `DrawRectangle` seguirá haciendo referencia a su versión en la clase derivada.

Pero en cuanto vuelva a compilar la aplicación con la nueva versión de `GraphicsClass`, recibirá una advertencia del compilador, CS0108. Esta advertencia le informa de que debe plantearse cómo quiere que el método `DrawRectangle` se comporte en la aplicación.

Si quiere que su método reemplace al nuevo método de clase base, use la palabra clave `override`:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

La palabra clave `override` se asegura de que los objetos derivados de `YourDerivedGraphicsClass` usen la versión de la clase derivada de `DrawRectangle`. Los objetos derivados de `YourDerivedGraphicsClass` todavía pueden acceder a la versión de clase base `DrawRectangle` mediante la palabra clave base:

```
base.DrawRectangle();
```

Si no quiere que el método reemplace al nuevo método de clase base, se aplican las consideraciones siguientes. Para evitar la confusión entre los dos métodos, puede cambiarle el nombre a su método. Esto puede ser un proceso lento y propenso a errores y no resultar práctico en algunos casos. Pero si el proyecto es relativamente pequeño, puede usar opciones de refactorización de Visual Studio para cambiar el nombre del método. Para obtener más información, vea Refactoring Classes and Types (Class Designer) (Refactorización de clases y tipos [Diseñador de clases]).

También puede evitar la advertencia mediante la palabra clave `new` en la definición de clase derivada:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Con la palabra clave `new` se indica al compilador que su definición oculta la definición contenida en la clase base. Éste es el comportamiento predeterminado.

Selección de método y reemplazo

Cuando se llama a un método en una clase, el compilador de C# selecciona el mejor método para llamar si hay más de uno compatible con la llamada, como cuando hay dos métodos con el mismo nombre y parámetros que son compatibles con el parámetro pasado. Los métodos siguientes serían compatibles:

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

Cuando se llama a `DoWork` en una instancia de `Derived`, el compilador de C# intentará en primer lugar que la llamada sea compatible con las versiones de `DoWork` declaradas originalmente en `Derived`. Los métodos de reemplazo no se consideran como declarados en una clase, son nuevas implementaciones de un método que se declara en una clase base. Solo si el compilador de C# no puede hacer coincidir la llamada al método con un método original en `Derived`, intentará hacer coincidir la llamada con un método reemplazado con el mismo nombre y parámetros compatibles. Por ejemplo:

```
int val = 5;
Derived d = new Derived();
d.DoWork(val);  // Calls DoWork(double).
```

Dado que la variable `val` se puede convertir implícitamente en un valor doble, el compilador de C# llama a `DoWork(double)` en lugar de a `DoWork(int)`. Hay dos maneras de evitarlo. En primer lugar, evite declarar nuevos métodos con el mismo nombre que los métodos virtuales. En segundo lugar, puede indicar al compilador de C# que llame al método virtual haciendo que busque la lista de métodos de clase base mediante la conversión de la instancia de `Derived` a `Base`. Como el método es virtual, se llamará a la implementación de `DoWork(int)` en `Derived`. Por ejemplo:

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

Para obtener otros ejemplos de `new` y `override`, vea Saber cuándo utilizar las palabras clave Override y New (Guía de programación de C#).

Vea también

- Guía de programación de C#
- Clases y structs
- Métodos
- Herencia

Saber cuándo utilizar las palabras clave Override y New (Guía de programación de C#)

18/03/2020 • 16 minutes to read • Edit Online

En C#, un método de una clase derivada puede tener el mismo nombre que un método de la clase base. Se puede especificar cómo interactúan los métodos mediante las palabras clave new y override. El modificador `override` *extiende* el método de clase base `virtual` y el modificador `new` *oculta* un método de clase base accesible. En los ejemplos de este tema se ilustra la diferencia.

En una aplicación de consola, declare las dos clases siguientes, `BaseClass` y `DerivedClass`. `DerivedClass` hereda de `BaseClass`.

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

En el método `Main`, declare las variables `bc`, `dc` y `bcdc`.

- `bc` es de tipo `BaseClass`, y su valor es de tipo `BaseClass`.
- `dc` es de tipo `DerivedClass`, y su valor es de tipo `DerivedClass`.
- `bcdc` es de tipo `BaseClass`, y su valor es de tipo `DerivedClass`. Esta es la variable a la que hay que prestar atención.

Dado que `bc` y `bcdc` tienen el tipo `BaseClass`, solo pueden tener acceso directo a `Method1`, a menos que se use la conversión. La variable `dc` puede tener acceso a `Method1` y `Method2`. Estas relaciones se muestran en el código siguiente.

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Después, agregue el método `Method2` siguiente a `BaseClass`. La firma de este método coincide con la firma del método `Method2` de `DerivedClass`.

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Dado que `BaseClass` ahora tiene un método `Method2`, se puede agregar una segunda instrucción de llamada para las variables de `BaseClass``bc` y `bcdc`, como se muestra en el código siguiente.

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

Al compilar el proyecto, verá que la adición del método `Method2` de `BaseClass` genera una advertencia. La advertencia indica que el método `Method2` de `DerivedClass` oculta el método `Method2` de `BaseClass`. Se recomienda usar la palabra clave `new` en la definición de `Method2` si se pretende provocar ese resultado. Como alternativa, se puede cambiar el nombre de uno de los métodos `Method2` para resolver la advertencia, pero eso no siempre resulta práctico.

Antes de agregar `new`, ejecute el programa para ver el resultado producido por las instrucciones adicionales que realizan la llamada. Se muestran los resultados siguientes.

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

La palabra clave `new` conserva las relaciones que generan ese resultado, pero se suprime la advertencia. Las variables de tipo `BaseClass` siguen teniendo acceso a los miembros de `BaseClass` y la variable de tipo

`DerivedClass` sigue teniendo acceso a los miembros de `DerivedClass` en primer lugar y, después, tiene en cuenta los miembros heredados de `BaseClass`.

Para suprimir la advertencia, agregue el modificador `new` a la definición de `Method2` en `DerivedClass`, como se muestra en el código siguiente. Se puede agregar el modificador antes o después de `public`.

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Vuelva a ejecutar el programa para comprobar que el resultado no ha cambiado. Compruebe también que ya no aparece la advertencia. Mediante el uso de `new`, afirma que es consciente de que el miembro que modifica oculta un miembro heredado de la clase base. Para más información sobre la ocultación de nombres a través de la herencia, vea new (Modificador, Referencia de C#).

Para contrastar este comportamiento con los efectos de usar `override`, agregue el método siguiente a `DerivedClass`. Se puede agregar el modificador `override` antes o después de `public`.

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Agregue el modificador `virtual` a la definición de `Method1` en `BaseClass`. Se puede agregar el modificador `virtual` antes o después de `public`.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Vuelva a ejecutar el proyecto. Observe especialmente las dos últimas líneas del resultado siguiente.

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

El uso del modificador `override` permite que `bcdc` tenga acceso al método `Method1` que se define en `DerivedClass`. Normalmente, es el comportamiento deseado en jerarquías de herencia. La intención es que los objetos que tienen valores que se crean a partir de la clase derivada usen los métodos que se definen en la clase derivada. Ese comportamiento se consigue mediante el uso de `override` para extender el método de clase base.

El código siguiente contiene el ejemplo completo.

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

En el ejemplo siguiente se muestra un comportamiento similar en un contexto diferente. El ejemplo define tres clases: una clase base denominada `Car` y dos clases que se derivan de ella, `ConvertibleCar` y `Minivan`. La clase base contiene un método `DescribeCar`. El método muestra una descripción básica de un automóvil y, después, llama a `ShowDetails` para proporcionar información adicional. Cada una de las tres clases define un método `ShowDetails`. El modificador `new` se usa para definir `ShowDetails` en la clase `ConvertibleCar`. El modificador `override` se usa para definir `ShowDetails` en la clase `Minivan`.

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

El ejemplo comprueba la versión de `ShowDetails` que se llama. El siguiente método, `TestCars1`, declara una instancia de cada clase y, después, llama a `DescribeCar` en cada instancia.

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` genera el siguiente resultado. Observe especialmente los resultados de `car2`, que probablemente no son los que se esperaban. El tipo de objeto es `ConvertibleCar`, pero `DescribeCar` no tiene acceso a la versión de `ShowDetails` que se define en la clase `ConvertibleCar` porque ese método se declara con el modificador `new`, no con el modificador `override`. Como resultado, un objeto `ConvertibleCar` muestra la misma descripción que un objeto `Car`. Compare los resultados de `car3`, que es un objeto `Minivan`. En este caso, el método `ShowDetails` que se declara en la clase `Minivan` invalida el método `ShowDetails` que se declara en la clase `Car`, y en la descripción que se muestra se describe una furgoneta.

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` crea una lista de objetos que tienen el tipo `Car`. Se crean instancias de los valores de los objetos desde las clases `Car`, `ConvertibleCar` y `Minivan`. `DescribeCar` se llama en cada elemento de la lista. En el código siguiente se muestra la definición de `TestCars2`.

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

Se muestra el siguiente resultado. Observe que es el mismo resultado mostrado por `TestCars1`. El método `ShowDetails` de la clase `ConvertibleCar` no se llama, independientemente de si el tipo de objeto es `ConvertibleCar`, como en `TestCars1`, o `Car` como en `TestCars2`. Por el contrario, `car3` llama al método `ShowDetails` desde la clase `Minivan` en ambos casos, independientemente de que tenga el tipo `Minivan` o `Car`.

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Los métodos `TestCars3` y `TestCars4` completan el ejemplo. Estos métodos llaman directamente a `ShowDetails`, primero desde los objetos declarados con el tipo `ConvertibleCar` y `Minivan` (`TestCars3`), y después desde los objetos declarados con el tipo `Car` (`TestCars4`). En el código siguiente se definen estos dos métodos.

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

Los métodos generan el siguiente resultado, que se corresponde a los resultados del primer ejemplo de este tema.

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

En el código siguiente se muestra el proyecto completo y sus resultados.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
```

```
            }
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars4
        // ----------
        // Standard transportation.
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
    // class also defines a ShowDetails method. The example tests which version of
    // ShowDetails is used, the base class method or the derived class method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
    // hides the base class method.
    class ConvertibleCar : Car
```

```
    {
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }

}
```

Vea también

- Guía de programación de C#
- Clases y structs
- Control de versiones con las palabras clave Override y New
- base
- abstract

Procedimiento para invalidar el método ToString (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Cada clase o struct de C# hereda implícitamente la clase Object. Por consiguiente, cada objeto de C# obtiene el método ToString, que devuelve una representación de cadena de ese objeto. Por ejemplo, todas las variables de tipo `int` tienen un método `ToString`, que las habilita para devolver su contenido como cadena:

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

Cuando cree una clase o struct personalizados, debe reemplazar el método ToString para proporcionar información sobre el tipo al código de cliente.

Para obtener información sobre cómo usar cadenas de formato y otros tipos de formato personalizado con el método `ToString`, vea Aplicar formato a tipos.

> **IMPORTANT**
> Cuando decida qué información va a proporcionar a través de este método, considere si la clase o struct se va a usar alguna vez en código que no sea de confianza. Asegúrese de que no proporciona información que pueda ser aprovechada por código malintencionado.

Para reemplazar el método `ToString` en una clase o struct:

1. Declare un método `ToString` con los modificadores y el tipo de valor devuelto siguientes:

   ```
   public override string ToString(){}
   ```

2. Implemente el método para que devuelva una cadena.

   En el ejemplo siguiente, se devuelve el nombre de la clase, además de los datos específicos de una instancia concreta de la clase.

   ```
   class Person
   {
       public string Name { get; set; }
       public int Age { get; set; }

       public override string ToString()
       {
           return "Person: " + Name + " " + Age;
       }
   }
   ```

   Se puede probar el método `ToString` tal y como se muestra en el siguiente ejemplo de código:

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

Vea también

- IFormattable
- Guía de programación de C#
- Clases y structs
- Cadenas
- string
- override
- virtual
- Aplicación de formato a tipos

Miembros (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

Las clases y structs tienen miembros que representan sus datos y comportamiento. Los miembros de una clase incluyen todos los miembros declarados en la clase, junto con todos los miembros (excepto constructores y finalizadores) declarados en todas las clases de su jerarquía de herencia. Los miembros privados de clases base se heredan en las clases derivadas, pero estas no pueden tener acceso a ellos.

En la tabla siguiente se enumeran los tipos de miembros que puede contener una clase o struct:

MEMBER	DESCRIPTION
Campos	Los campos son variables declaradas en el ámbito de clase. Un campo puede ser un tipo numérico integrado o una instancia de otra clase. Por ejemplo, una clase de calendario puede tener un campo con la fecha actual.
Constantes	Las constantes son campos cuyo valor se establece en tiempo de compilación y no se puede cambiar.
Propiedades	Las propiedades son métodos de una clase a los que se obtiene acceso como si fueran campos de esa clase. Una propiedad puede proporcionar protección a un campo de clase con el fin de evitar que se cambie sin el conocimiento del objeto.
Métodos	Los métodos definen las acciones que una clase puede realizar. Los métodos pueden aceptar parámetros que proporcionan datos de entrada y devolver datos de salida a través de parámetros. Los métodos también pueden devolver un valor directamente, sin usar ningún parámetro.
Eventos	Los eventos proporcionan a otros objetos notificaciones sobre lo que ocurre, como clics en botones o la realización correcta de un método. Los eventos se definen y desencadenan mediante delegados.
Operadores	Los operadores sobrecargados se consideran miembros de tipo. Si se sobrecarga un operador, se define como método estático público en un tipo. Para obtener más información, vea Sobrecarga de operadores.
Indizadores	Los indizadores permiten indizar un objeto de manera similar a como se hace con las matrices.
Constructores	Los constructores son métodos a los que se llama cuando el objeto se crea por primera vez. Se usan a menudo para inicializar los datos de un objeto.
Finalizadores	En C#, los finalizadores se usan en raras ocasiones. Son métodos a los que llama el motor de ejecución del runtime cuando el objeto está a punto de quitarse de la memoria. Generalmente se utilizan para asegurarse de que los recursos que se deben liberar se controlan apropiadamente.

MEMBER	DESCRIPTION
Tipos anidados	Los tipos anidados son tipos declarados dentro de otro tipo. Los tipos anidados se usan a menudo para describir objetos utilizados únicamente por los tipos que los contienen.

Vea también

- Guía de programación de C#
- Clases

Clases y miembros de clase abstractos y sellados (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

La palabra clave abstract permite crear clases y miembros class que están incompletos y se deben implementar en una clase derivada.

La palabra clave sealed permite impedir la herencia de una clase o de ciertos miembros de clase marcados previamente como virtual.

Clases y miembros de clase abstractos

Las clases se pueden declarar como abstractas si se incluye la palabra clave `abstract` antes de la definición de clase. Por ejemplo:

```
public abstract class A
{
    // Class members here.
}
```

No se pueden crear instancias de una clase abstracta. El propósito de una clase abstracta es proporcionar una definición común de una clase base que múltiples clases derivadas pueden compartir. Por ejemplo, una biblioteca de clases puede definir una clase abstracta que se utiliza como parámetro para muchas de sus funciones y solicitar a los programadores que utilizan esa biblioteca que proporcionen su propia implementación de la clase mediante la creación de una clase derivada.

Las clases abstractas también pueden definir métodos abstractos. Esto se consigue agregando la palabra clave `abstract` antes del tipo de valor que devuelve el método. Por ejemplo:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Los métodos abstractos no tienen ninguna implementación, de modo que la definición de método va seguida por un punto y coma en lugar de un bloque de método normal. Las clases derivadas de la clase abstracta deben implementar todos los métodos abstractos. Cuando una clase abstracta hereda un método virtual de una clase base, la clase abstracta puede reemplazar el método virtual con un método abstracto. Por ejemplo:

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

Si un método `virtual` se declara como `abstract`, sigue siendo virtual para cualquier clase que herede de la clase abstracta. Una clase que hereda un método abstracto no puede tener acceso a la implementación original del método: en el ejemplo anterior, `DoWork` en la clase F no puede llamar a `DoWork` en la clase D. De esta manera, una clase abstracta puede exigir a las clases derivadas que proporcionen nuevas implementaciones de método para los métodos virtuales.

Clases y miembros de clase sellados

Las clases se pueden declarar como selladas si se incluye la palabra clave `sealed` antes de la definición de clase. Por ejemplo:

```
public sealed class D
{
    // Class members here.
}
```

Una clase sellada no se puede utilizar como clase base. Por esta razón, tampoco puede ser una clase abstracta. Las clases selladas evitan la derivación. Puesto que nunca se pueden utilizar como clase base, algunas optimizaciones en tiempo de ejecución pueden hacer que sea un poco más rápido llamar a miembros de clase sellada.

Un método, indizador, propiedad o evento de una clase derivada que reemplaza a un miembro virtual de la clase base puede declarar ese miembro como sellado. Esto niega el aspecto virtual del miembro para cualquier clase derivada adicional. Esto se logra colocando la palabra clave `sealed` antes de la palabra clave override en la declaración del miembro de clase. Por ejemplo:

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

Vea también

- Guía de programación de C#
- Clases y structs

- Herencia
- Métodos
- Campos
- Procedimiento para definir propiedades abstractas

Clases estáticas y sus miembros (Guía de programación de C#)

29/06/2020 • 10 minutes to read • Edit Online

Una clase estática es básicamente lo mismo que una clase no estática, con la diferencia de que no se pueden crear instancias de una clase estática. En otras palabras, no puede usar el operador new para crear una variable del tipo de clase. Dado que no hay ninguna variable de instancia, para tener acceso a los miembros de una clase estática, debe usar el nombre de la clase. Por ejemplo, si tiene una clase estática denominada `UtilityClass` que tiene un método estático público denominado `MethodA`, llame al método tal como se muestra en el ejemplo siguiente:

```
UtilityClass.MethodA();
```

Es posible usar una clase estática como un contenedor adecuado para conjuntos de métodos que solo funcionan en parámetros de entrada y que no tienen que obtener ni establecer campos de instancias internas. Por ejemplo, en la biblioteca de clases .NET, la clase estática System.Math contiene métodos que realizan operaciones matemáticas, sin ningún requisito para almacenar o recuperar datos que sean únicos de una instancia concreta de la clase Math. Es decir, para aplicar los miembros de la clase, debe especificar el nombre de clase y el nombre de método, como se muestra en el ejemplo siguiente.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

Como sucede con todos los tipos de clase, el entorno de ejecución de .NET carga la información de tipo para una clase estática cuando se carga el programa que hace referencia a la clase. El programa no puede especificar exactamente cuándo se carga la clase, pero existe la garantía de que se cargará, de que sus campos se inicializarán y de que se llamará a su constructor estático antes de que se haga referencia a la clase por primera vez en el programa. Solo se llama una vez a un constructor estático, y una clase estática permanece en memoria durante la vigencia del dominio de aplicación en el que reside el programa.

> **NOTE**
>
> Para crear una clase no estática que solo permita la creación de una instancia de sí misma, vea Implementing Singleton in C# (Implementar un singleton en C#).

La siguiente lista contiene las características principales de una clase estática:

- Solo contiene miembros estáticos.
- No se pueden crear instancias de ella.
- Está sellada.

- No puede contener constructores de instancias.

Por lo tanto, crear una clase estática es básicamente lo mismo que crear una clase que contiene solo miembros estáticos y un constructor privado. Un constructor privado impide que se creen instancias de la clase. La ventaja de usar una clase estática es que el compilador puede comprobar que no se agregue accidentalmente ningún miembro de instancia. El compilador garantizará que no se creen instancias de esta clase.

Las clases estáticas están selladas y, por lo tanto, no pueden heredarse. No pueden heredar de ninguna clase excepto Object. Las clases estáticas no pueden contener un constructor de instancia, aunque sí un constructor estático. Las clases no estáticas también deben definir un constructor estático si la clase contiene miembros estáticos que requieren inicialización no trivial. Para obtener más información, vea Constructores estáticos.

Ejemplo

A continuación se muestra un ejemplo de una clase estática que contiene dos métodos que convierten la temperatura de grados Celsius a grados Fahrenheit y viceversa:

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
```

```
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

Miembros estáticos

Una clase no estática puede contener métodos, campos, propiedades o eventos estáticos. El miembro estático es invocable en una clase, incluso si no se ha creado ninguna instancia de la clase. Siempre se tiene acceso al miembro estático con el nombre de clase, no con el nombre de instancia. Solo existe una copia de un miembro estático, independientemente del número de instancias de la clase que se creen. Los métodos y las propiedades estáticos no pueden acceder a campos y eventos no estáticos en su tipo contenedor, ni tampoco a una variable de instancia de un objeto a menos que se pase explícitamente en un parámetro de método.

Es más habitual declarar una clase no estática con algunos miembros estáticos que declarar toda una clase como estática. Dos usos habituales de los campos estáticos son llevar la cuenta del número de objetos de los que se han creado instancias y almacenar un valor que se debe compartir entre todas las instancias.

Los métodos estáticos se pueden sobrecargar pero no invalidar, ya que pertenecen a la clase y no a una instancia de la clase.

Aunque un campo no se puede declarar como `static const`, el campo const es básicamente estático en su comportamiento. Pertenece al tipo, no a las instancias del tipo. Por lo tanto, se puede acceder a campos `const` mediante la misma notación `ClassName.MemberName` que se usa para los campos estáticos. No se requiere ninguna instancia de objeto.

C# no admite variables locales estáticas (es decir, variables que se declaran en el ámbito del método).

Para declarar miembros de clases estáticas, use la palabra clave `static` antes del tipo de valor devuelto del miembro, como se muestra en el ejemplo siguiente:

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }
    public static void Drive() { }
    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

Los miembros estáticos se inicializan antes de que se obtenga acceso por primera vez al miembro estático y antes de que se llame al constructor estático, en caso de haberlo. Para tener acceso a un miembro de clase estática, use el nombre de la clase en lugar de un nombre de variable para especificar la ubicación del miembro, como se muestra en el ejemplo siguiente:

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

Si la clase contiene campos estáticos, proporcione un constructor estático que los inicialice al cargar la clase.

Una llamada a un método estático genera una instrucción de llamada en Lenguaje Intermedio de Microsoft (MSIL), mientras que una llamada a un método de instancia genera una instrucción `callvirt`, que además busca referencias a un objeto NULL, pero la mayoría de las veces la diferencia de rendimiento entre las dos no es significativo.

Especificación del lenguaje C#

Para más información, vea las secciones sobre clases estáticas y sobre miembros de instancia y estáticos de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- static
- Clases
- class
- Constructores estáticos
- Constructores de instancias

Modificadores de acceso (Guía de programación de C#)

18/03/2020 • 8 minutes to read • Edit Online

Todos los tipos y miembros de tipo tienen un nivel de accesibilidad. El nivel de accesibilidad controla si se pueden usar desde otro código del ensamblado u otros ensamblados. Use los modificadores de acceso siguientes para especificar la accesibilidad de un tipo o miembro cuando lo declare:

- public: Puede obtener acceso al tipo o miembro cualquier otro código del mismo ensamblado o de otro ensamblado que haga referencia a éste.
- private: solamente el código de la misma `class` o `struct` puede acceder al tipo o miembro.
- protected: solamente el código de la misma `class`, o bien de una `class` derivada de esa `class`, puede acceder al tipo o miembro.
- internal: Puede obtener acceso al tipo o miembro cualquier código del mismo ensamblado, pero no de un ensamblado distinto.
- protected internal: cualquier código del ensamblado en el que se ha declarado, o desde una `class` derivada de otro ensamblado, puede acceder al tipo o miembro.
- private protected: el código de la misma `class`, o de un tipo derivado de esa `class`, puede acceder al tipo o miembro solo dentro de su ensamblado de declaración.

En los ejemplos siguientes se muestra cómo especificar modificadores de acceso en un tipo y miembro:

```
public class Bicycle
{
    public void Pedal() { }
}
```

No todos los modificadores de acceso son válidos para todos los tipos o miembros de todos los contextos. En algunos casos, la accesibilidad de un miembro de tipo está restringida por la accesibilidad de su tipo contenedor.

Accesibilidad de clases y estructuras

Las clases y estructuras que se declaran directamente en un espacio de nombres (es decir, que no están anidadas en otras clases o estructuras) pueden ser `public` o `internal`. Si no se especifica ningún modificador de acceso, el valor predeterminado es `Internal`.

Los miembros de estructura, incluidas las clases y las estructuras anidadas, se pueden declarar como `public`, `internal` o `private`. Los miembros de clase, incluidas las clases y las estructuras anidadas, pueden ser `public`, `protected internal`, `protected`, `internal`, `private protected` o `private`. Los miembros de clase y de estructura, incluidas las clases y estructuras anidadas, tienen acceso `private` de forma predeterminada. Los tipos anidados privados no son accesibles desde fuera del tipo contenedor.

Las clases derivadas no pueden tener mayor accesibilidad que sus tipos base. No se puede declarar una clase pública `B` que derive de una clase interna `A`. Si se permitiera, convertiría `A` en público, porque todos los miembros `protected` o `internal` de `A` son accesibles desde la clase derivada.

Puede habilitar otros ensamblados concretos para acceder a los tipos internos mediante `InternalsVisibleToAttribute`. Para más información, vea Ensamblados de confianza.

Accesibilidad de miembros de clase y estructura

Los miembros de clase (incluidas las clases y las estructuras anidadas) se pueden declarar con cualquiera de los seis tipos de acceso. Los miembros de estructura no se pueden declarar como `protected`, `protected internal` o `private protected` porque las estructuras no admiten la herencia.

Normalmente, la accesibilidad de un miembro no es mayor que la del tipo que lo contiene. Pero un miembro `public` de una clase interna podría ser accesible desde fuera del ensamblado si el miembro implementa los métodos de interfaz o invalida los métodos virtuales definidos en una clase base pública.

El tipo de cualquier miembro que sea un campo, propiedad o evento debe ser al menos tan accesible como el propio miembro. Del mismo modo, el tipo devuelto y los tipos de parámetro de cualquier método, indizador o delegado deben ser al menos tan accesibles como el propio miembro. Por ejemplo, no puede tener un método `public` `M` que devuelva una clase `C` a menos que `C` también sea `public`. Del mismo modo, no puede tener una propiedad `protected` de tipo `A` si `A` se declara como `private`.

Los operadores definidos por el usuario siempre se deben declarar como `public` y `static`. Para obtener más información, vea Sobrecarga de operadores.

Los finalizadores no pueden tener modificadores de accesibilidad.

Para establecer el nivel de acceso de un miembro de `class` o `struct`, agregue la palabra clave adecuada a la declaración de miembro, como se muestra en el ejemplo siguiente.

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return wheels; }
    }
}
```

Otros tipos

Las interfaces declaradas directamente en un espacio de nombres pueden ser `public` o `internal` y, al igual que las clases y las estructuras, su valor predeterminado es el acceso `internal`. Los miembros de interfaz son `public` de manera predeterminada porque el propósito de una interfaz es permitir que otros tipos accedan a una clase o estructura. Las declaraciones de miembros de interfaz pueden incluir cualquier modificador de acceso. Esto es muy útil para que los métodos estáticos proporcionen implementaciones comunes necesarias para todos los implementadores de una clase.

Los miembros de enumeración siempre son `public` y no se les puede aplicar ningún modificador de acceso.

Los delegados se comportan como las clases y las estructuras. De forma predeterminada, tienen acceso `internal` cuando se declaran directamente en un espacio de nombres y acceso `private` cuando están anidados.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Clases y structs
- Interfaces
- private
- public
- internal
- protected
- protected internal
- private protected
- class
- struct
- interface

Campos (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

Un *campo* es una variable de cualquier tipo que se declara directamente en una clase o struct. Los campos son *miembros* de su tipo contenedor.

Una clase o struct puede tener campos de instancia, campos estáticos o ambos. Los campos de instancia son específicos de una instancia de un tipo. Si tiene una clase T, con un campo de instancia F, puede crear dos objetos de tipo T y modificar el valor de F en cada objeto sin afectar el valor del otro objeto. Por el contrario, un campo estático pertenece a la propia clase y se comparte entre todas las instancias de esa clase. Solo puede tener acceso al campo estático mediante el nombre de clase. Si obtiene acceso al campo estático mediante un nombre de instancia, obtendrá el error en tiempo de compilación CS0176.

Por lo general, solo se deben usar campos para las variables que tienen accesibilidad privada o protegida. Los datos que la clase expone al código de cliente deben proporcionarse a través de métodos, propiedades e indizadores. Mediante estas construcciones para el acceso indirecto a los campos internos, se puede proteger de los valores de entrada no válidos. Un campo privado que almacena los datos expuestos por una propiedad pública se denomina *memoria auxiliar* o *campo de respaldo*.

Los campos almacenan habitualmente los datos que deben ser accesibles para más de un método de clase y que deben almacenarse durante más tiempo de lo que dura un único método. Por ejemplo, es posible que una clase que representa una fecha de calendario tenga tres campos enteros: uno para el mes, otro para el día y otro para el año. Las variables que no se usan fuera del ámbito de un único método se deben declarar como *variables locales* dentro del campo del método.

Los campos se declaran en el bloque de clase especificando el nivel de acceso del campo, seguido por el tipo del campo, seguido por el nombre del campo. Por ejemplo:

```
public class CalendarEntry
{
    // private field
    private DateTime date;

    // public field (Generally not recommended.)
    public string day;

    // Public property exposes date field safely.
    public DateTime Date
    {
        get
        {
            return date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException();
            }
        }
    }

    // Public method also exposes date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt != null && dt.Ticks < date.Ticks)
        {
            return date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}
```

Para obtener acceso a un campo en un objeto, agregue un punto después del nombre de objeto, seguido del nombre del campo, como en `objectname.fieldname`. Por ejemplo:

```
CalendarEntry birthday = new CalendarEntry();
birthday.day = "Saturday";
```

Se puede proporcionar un valor inicial a un campo mediante el operador de asignación cuando se declara el campo. Para asignar automáticamente el campo `day` a `"Monday"`, por ejemplo, se declararía `day` como en el ejemplo siguiente:

```
public class CalendarDateWithInitialization
{
    public string day = "Monday";
    //...
}
```

Los campos se inicializan inmediatamente antes de que se llame al constructor para la instancia del objeto. Si el constructor asigna el valor de un campo, sobrescribirá cualquier valor dado durante la declaración del campo. Para obtener más información, vea Utilizar constructores.

> **NOTE**
> Un inicializador de campo no puede hacer referencia a otros campos de instancia.

Se pueden marcar campos como público, privado, protegido, interno, protegido interno o privado protegido. Estos modificadores de acceso definen cómo los usuarios de la clase pueden obtener acceso a los campos. Para obtener más información, vea Modificadores de acceso.

Opcionalmente, un campo se puede declarar como static. Esto hace que el campo esté disponible para los llamadores en cualquier momento, aunque no exista ninguna instancia de la clase. Para obtener más información, consulte Clases estáticas y sus miembros.

Se puede declarar un campo como readonly. Solamente se puede asignar un valor a un campo de solo lectura durante la inicialización o en un constructor. Un campo `static readonly` es muy similar a una constante, salvo que el compilador de C# no tiene acceso al valor de un campo estático de solo lectura en tiempo de compilación, solo en tiempo de ejecución. Para obtener más información, vea Constants (Constantes).

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Clases y structs
- Utilizar constructores
- Herencia
- Modificadores de acceso
- Clases y miembros de clase abstractos y sellados

Constantes (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Las constantes son valores inmutables que se conocen en tiempo de compilación y que no cambian durante la vida del programa. Las constantes se declaran con el modificador const. Solo los tipos integrados de C# (excluido System.Object) pueden declararse como `const`. Los tipos definidos por el usuario, incluidas las clases, las estructuras y las matrices, no pueden ser `const`. Use el modificador readonly para crear una clase, una estructura o una matriz que se inicialice una vez en tiempo de ejecución (por ejemplo, en un constructor) y que posteriormente no se pueda cambiar.

C# no admite métodos, propiedades ni eventos `const`.

El tipo enum permite definir constantes con nombre para los tipos integrados enteros (por ejemplo, `int`, `uint`, `long`, etc.). Para más información, vea enum.

Las constantes se deben inicializar al declararse. Por ejemplo:

```
class Calendar1
{
    public const int Months = 12;
}
```

En este ejemplo la constante `months` siempre es 12 y ni siquiera la propia clase la puede cambiar. De hecho, cuando el compilador detecta un identificador de constante en el código fuente de C# (por ejemplo, `months`), sustituye directamente el valor literal en el código de lenguaje intermedio (IL) que genera. Dado que no hay ninguna dirección de variable asociada a una constante en tiempo de ejecución, no se pueden pasar los campos `const` por referencia ni pueden aparecer como un valor L en una expresión.

> **NOTE**
> Tenga cuidado al hacer referencia a valores de constante definidos en otro código como archivos DLL. Si una nueva versión del archivo DLL define un nuevo valor para la constante, el programa conservará el valor literal anterior hasta que se vuelva a compilar con la versión nueva.

Se pueden declarar varias constantes del mismo tipo a la vez, por ejemplo:

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

La expresión que se usa para inicializar una constante puede hacer referencia a otra constante si no crea una referencia circular. Por ejemplo:

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Las constantes pueden marcarse como públicas, privadas, protegidas, internas, protegidas internaso privadas protegidas. Estos modificadores de acceso definen cómo los usuarios de la clase pueden acceder a la constante. Para obtener más información, vea Modificadores de acceso.

A las constantes se accede como si fueran campos estáticos porque el valor de la constante es el mismo para todas las instancias del tipo. No use la palabra clave `static` para declararlas. Las expresiones que no están en la clase que define la constante deben usar el nombre de la clase, un punto y el nombre de la constante para acceder a ella. Por ejemplo:

```
int birthstones = Calendar.Months;
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Clases y structs
- Propiedades
- Tipos
- readonly
- Immutability in C# Part One: Kinds of Immutability (Inmutabilidad en C# (primera parte): tipos de inmutabilidad)

Procedimiento para definir propiedades abstractas (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

En el ejemplo siguiente se muestra cómo definir las propiedades abstract. Una declaración de propiedad abstracta no proporciona una implementación de los descriptores de acceso de propiedad, declara que la clase admite propiedades, pero deja la implementación del descriptor de acceso a las clases derivadas. En el ejemplo siguiente se muestra cómo implementar las propiedades abstractas heredadas de una clase base.

Este ejemplo consta de tres archivos, cada uno de los cuales se compila individualmente y se hace referencia a su ensamblado resultante mediante la siguiente compilación:

- abstractshape.cs: la clase `Shape` que contiene una propiedad `Area` abstracta.
- shapes.cs: las subclases de la clase `Shape`.
- shapetest.cs: un programa de prueba para mostrar las áreas de algunos objetos derivados de `Shape`.

Para compilar el ejemplo, use el siguiente comando:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

Esto creará el archivo ejecutable shapetest.exe.

Ejemplo

Este archivo declara la clase `Shape` que contiene la propiedad `Area` del tipo `double`.

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Los modificadores de la propiedad se colocan en la propia declaración de propiedad. Por ejemplo:

  ```
  public abstract double Area
  ```

- Al declarar una propiedad abstracta (como `Area` en este ejemplo), simplemente indica qué descriptores de acceso de propiedad están disponibles, pero no los implementa. En este ejemplo, solo está disponible un descriptor de acceso get, por lo que la propiedad es de solo lectura.

Ejemplo

En el siguiente código se muestran tres subclases de `Shape` y cómo invalidan la propiedad `Area` para proporcionar su propia implementación.

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

Ejemplo

En el siguiente código se muestra un programa de prueba que crea un número de objetos derivados de `Shape` e imprime sus áreas.

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

Vea también

- Guía de programación de C#
- Clases y structs
- Clases y miembros de clase abstractos y sellados
- Propiedades

Definición de constantes en C#

18/03/2020 • 2 minutes to read • Edit Online

Las constantes son campos cuyos valores se establecen en tiempo de compilación y nunca se pueden cambiar. Use constantes para proporcionar nombres descriptivos en lugar de literales numéricos ("números mágicos") para valores especiales.

> **NOTE**
>
> En C#, la directiva del preprocesador #define no puede usarse para definir constantes en la manera que se usa normalmente en C y C++.

Para definir valores constantes de tipos enteros (`int`, `byte` y así sucesivamente) use un tipo enumerado. Para más información, vea enum.

Para definir constantes no enteras, un enfoque es agruparlas en una única clase estática denominada `Constants`. Esto necesitará que todas las referencias a las constantes vayan precedidas por el nombre de clase, como se muestra en el ejemplo siguiente.

Ejemplo

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}
class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
    }
}
```

El uso del calificador de nombre de clase ayuda a garantizar que usted y otros usuarios que usan la constante comprenden que es una constante y que no puede modificarse.

Vea también

- Clases y structs

Propiedades (Guía de programación de C#)

18/03/2020 • 9 minutes to read • Edit Online

Una propiedad es un miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo privado. Las propiedades se pueden usar como si fueran miembros de datos públicos, pero en realidad son métodos especiales denominados *descriptores de acceso*. Esto permite acceder fácilmente a los datos a la vez que proporciona la seguridad y la flexibilidad de los métodos.

Información general sobre propiedades

- Las propiedades permiten que una clase exponga una manera pública de obtener y establecer valores, a la vez que se oculta el código de implementación o verificación.
- Para devolver el valor de la propiedad se usa un descriptor de acceso de propiedad get, mientras que para asignar un nuevo valor se emplea un descriptor de acceso de propiedad set. Estos descriptores de acceso pueden tener diferentes niveles de acceso. Para más información, vea Restringir la accesibilidad del descriptor de acceso.
- La palabra clave value se usa para definir el valor que va a asignar el descriptor de acceso `set`.
- Las propiedades pueden ser *de lectura y escritura* (en ambos casos tienen un descriptor de acceso `get` y `set`), *de solo lectura* (tienen un descriptor de acceso `get`, pero no `set`) o *de solo escritura* (tienen un descriptor de acceso `set`, pero no `get`). Las propiedades de solo escritura son poco frecuentes y se suelen usar para restringir el acceso a datos confidenciales.
- Las propiedades simples que no necesitan ningún código de descriptor de acceso personalizado se pueden implementar como definiciones de cuerpos de expresión o como propiedades implementadas automáticamente.

Propiedades con campos de respaldo

Un patrón básico para implementar una propiedad conlleva el uso de un campo de respaldo privado para establecer y recuperar el valor de la propiedad. El descriptor de acceso `get` devuelve el valor del campo privado y el descriptor de acceso `set` puede realizar alguna validación de datos antes de asignar un valor al campo privado. Ambos descriptores de acceso además pueden realizar algún cálculo o conversión en los datos antes de que se almacenen o se devuelvan.

En el ejemplo siguiente se muestra este patrón. En este ejemplo, la clase `TimePeriod` representa un intervalo de tiempo. Internamente, la clase almacena el intervalo de tiempo en segundos en un campo privado denominado `_seconds`. Una propiedad de lectura y escritura denominada `Hours` permite al cliente especificar el intervalo de tiempo en horas. Los descriptores de acceso `get` y `set` realizan la conversión necesaria entre horas y segundos. Además, el descriptor de acceso `set` valida los datos e inicia una excepción ArgumentOutOfRangeException si el número de horas no es válido.

```
using System;

class TimePeriod
{
   private double _seconds;

   public double Hours
   {
       get { return _seconds / 3600; }
       set {
          if (value < 0 || value > 24)
             throw new ArgumentOutOfRangeException(
                   $"{nameof(value)} must be between 0 and 24.");

          _seconds = value * 3600;
       }
   }
}

class Program
{
   static void Main()
   {
       TimePeriod t = new TimePeriod();
       // The property assignment causes the 'set' accessor to be called.
       t.Hours = 24;

       // Retrieving the property causes the 'get' accessor to be called.
       Console.WriteLine($"Time in hours: {t.Hours}");
   }
}
// The example displays the following output:
//    Time in hours: 24
```

Definiciones de cuerpos de expresión

Los descriptores de acceso de propiedad suelen constar de instrucciones de una sola línea que simplemente asignan o devuelven el resultado de una expresión. Puede implementar estas propiedades como miembros con forma de expresión. Las definiciones de cuerpos de expresión constan del símbolo `=>` seguido de la expresión que se va a asignar a la propiedad o a recuperar de ella.

A partir de C# 6, las propiedades de solo lectura pueden implementar el descriptor de acceso `get` como miembro con forma de expresión. En este caso, no se usan ni la palabra clave del descriptor de acceso `get` ni la palabra clave `return`. En el ejemplo siguiente se implementa la propiedad de solo lectura `Name` como miembro con forma de expresión.

```
using System;

public class Person
{
   private string _firstName;
   private string _lastName;

   public Person(string first, string last)
   {
      _firstName = first;
      _lastName = last;
   }

   public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
   public static void Main()
   {
      var person = new Person("Magnus", "Hedlund");
      Console.WriteLine(person.Name);
   }
}
// The example displays the following output:
//      Magnus Hedlund
```

A partir de C# 7.0, los descriptores de acceso `get` y `set` se pueden implementar como miembros con forma de expresión. En este caso, las palabras clave `get` y `set` deben estar presentes. En el ejemplo siguiente se muestra el uso de definiciones de cuerpos de expresión para ambos descriptores de acceso. Observe que no se usa la palabra clave `return` con el descriptor de acceso `get`.

```
using System;

public class SaleItem
{
   string _name;
   decimal _cost;

   public SaleItem(string name, decimal cost)
   {
      _name = name;
      _cost = cost;
   }

   public string Name
   {
      get => _name;
      set => _name = value;
   }

   public decimal Price
   {
      get => _cost;
      set => _cost = value;
   }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem("Shoes", 19.95m);
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Propiedades implementadas automáticamente

En algunos casos, los descriptores de acceso de propiedad `get` y `set` simplemente asignan un valor a un campo de respaldo o recuperan un valor de él sin incluir ninguna lógica adicional. Mediante las propiedades implementadas automáticamente, puede simplificar el código y conseguir que el compilador de C# le proporcione el campo de respaldo de forma transparente.

Si una propiedad tiene un descriptor de acceso `get` y `set`, ambos deben ser implementados automáticamente. Una propiedad implementada automáticamente se define mediante las palabras clave `get` y `set` sin proporcionar ninguna implementación. El ejemplo siguiente repite el anterior, salvo que `Name` y `Price` son propiedades implementadas automáticamente. Tenga en cuenta que en el ejemplo también se quita el constructor parametrizado, por lo que los objetos `SaleItem` ahora se inicializan con una llamada al constructor sin parámetros y un inicializador de objeto.

```
using System;

public class SaleItem
{
   public string Name
   { get; set; }

   public decimal Price
   { get; set; }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Secciones relacionadas

- Utilizar propiedades
- Propiedades de interfaz
- Comparación entre propiedades e indizadores
- Restringir la accesibilidad del descriptor de acceso
- Propiedades autoimplementadas

Especificación del lenguaje C#

Para obtener más información, vea la sección Propiedades de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Utilizar propiedades
- Indizadores
- get (Palabra clave)
- set (palabra clave)

Utilizar propiedades (Guía de programación de C#)

18/03/2020 • 14 minutes to read • Edit Online

Las propiedades combinan aspectos de los campos y los métodos. Para el usuario de un objeto, una propiedad que parece un campo, el acceso a la propiedad necesita la misma sintaxis. Para el implementador de una clase, una propiedad es uno o dos bloques de código que representa un descriptor de acceso get o un descriptor de acceso set. El bloque de código del descriptor de acceso `get` se ejecuta cuando se lee la propiedad; el bloque de código del descriptor de acceso `set` se ejecuta cuando se asigna un nuevo valor a la propiedad. Una propiedad sin un descriptor de acceso `set` se considera de solo lectura. Una propiedad sin un descriptor de acceso `get` se considera de solo escritura. Una propiedad que tiene ambos descriptores de acceso es de lectura y escritura.

A diferencia de los campos, las propiedades no se clasifican como variables. Por lo tanto, no puede pasar una propiedad como un parámetro ref u out.

Las propiedades tienen muchos usos: pueden validar datos antes de permitir un cambio; pueden exponer claramente datos en una clase donde esos datos se recuperan realmente de otros orígenes, como una base de datos; pueden realizar una acción cuando los datos se cambian, como generar un evento, o cambiar el valor de otros campos.

Las propiedades se declaran en el bloque de clase especificando el nivel de acceso del campo, seguido del tipo de la propiedad, seguido del nombre de la propiedad y seguido de un bloque de código que declara un descriptor de acceso `get` o un descriptor de acceso `set`. Por ejemplo:

```
public class Date
{
    private int _month = 7;  // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

En este ejemplo, `Month` se declara como una propiedad, de manera que el descriptor de acceso `set` pueda estar seguro de que el valor `Month` se establece entre 1 y 12. La propiedad `Month` usa un campo privado para realizar un seguimiento del valor actual. A menudo, a la ubicación real de los datos de una propiedad se le conoce como la "memoria auxiliar" de la propiedad. Esto es común para las propiedades que usan campos privados como una memoria auxiliar. El campo se marca como privado para asegurarse de que solo puede cambiarse llamando a la propiedad. Para obtener más información sobre las restricciones de acceso público y privado, vea Modificadores de acceso.

Las propiedades implementadas automáticamente proporcionan una sintaxis simplificada para las declaraciones de propiedad simples. Para obtener más información, vea Auto-Implemented Properties (Propiedades implementadas automáticamente).

El descriptor de acceso get

El cuerpo del descriptor de acceso `get` se parece al de un método. Debe devolver un valor del tipo de propiedad. La ejecución del descriptor de acceso `get` es equivalente a la lectura del valor del campo. Por ejemplo, cuando se devuelve la variable privada del descriptor de acceso `get` y se habilitan las optimizaciones, la llamada al método de descriptor de acceso `get` se inserta mediante el compilador, de manera que no existe ninguna sobrecarga de llamada al método. En cambio, un método de descriptor de acceso `get` virtual no puede insertarse porque el compilador no conoce en tiempo de compilación a qué método puede llamarse realmente en tiempo de ejecución. A continuación se muestra un descriptor de acceso `get` que devuelve el valor de un campo privado `_name`:

```
class Person
{
    private string _name;  // the name field
    public string Name => _name;     // the Name property
}
```

Cuando hace referencia a la propiedad, excepto como el destino de una asignación, el descriptor de acceso `get` se invoca para leer el valor de la propiedad. Por ejemplo:

```
Person person = new Person();
//...

System.Console.Write(person.Name);  // the get accessor is invoked here
```

El descriptor de acceso `get` debe finalizar en una instrucción return o throw, y el control no puede salir del cuerpo del descriptor de acceso.

Cambiar el estado del objeto mediante el descriptor de acceso `get` es un estilo de programación incorrecto. Por ejemplo, el siguiente descriptor de acceso produce el efecto secundario de cambiar el estado del objeto cada vez que se tiene acceso al campo `_number`.

```
private int _number;
public int Number => _number++; // Don't do this
```

El descriptor de acceso `get` puede usarse para devolver el valor de campo o para calcularlo y devolverlo. Por ejemplo:

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

En el segmento de código anterior, si no asigna un valor a la propiedad `Name`, devolverá el valor `NA`.

El descriptor de acceso set

El descriptor de acceso `set` es similar a un método cuyo tipo de valor devuelto es void. Usa un parámetro implícito denominado `value`, cuyo tipo es el tipo de la propiedad. En el siguiente ejemplo, se agrega un descriptor de acceso `set` a la propiedad `Name`:

```
class Person
{
    private string _name;  // the name field
    public string Name    // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

Cuando asigna un valor a la propiedad, el descriptor de acceso `set` se invoca mediante un argumento que proporciona el valor nuevo. Por ejemplo:

```
Person person = new Person();
person.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(person.Name);  // the get accessor is invoked here
```

Es un error usar el nombre de parámetro implícito, `value`, para una declaración de variable local en el descriptor de acceso `set`.

Comentarios

Las propiedades pueden marcarse como `public`, `private`, `protected`, `internal`, `protected internal` o `private protected`. Estos modificadores de acceso definen cómo los usuarios de la clase pueden obtener acceso a la propiedad. Los descriptores de acceso `get` y `set` para la misma propiedad pueden tener diferentes modificadores de acceso. Por ejemplo, `get` puede ser `public` para permitir el acceso de solo lectura desde el exterior del tipo, y `set` puede ser `private` o `protected`. Para obtener más información, vea Modificadores de acceso.

Una propiedad puede declararse como una propiedad estática mediante la palabra clave `static`. Esto hace que la propiedad esté disponible para los autores de la llamada en cualquier momento, aunque no exista ninguna instancia de la clase. Para obtener más información, consulte Clases estáticas y sus miembros.

Una propiedad puede marcarse como una propiedad virtual mediante la palabra clave virtual. Esto permite que las clases derivadas invaliden el comportamiento de la propiedad mediante la palabra clave override. Para obtener más información sobre estas opciones, vea Herencia.

Una propiedad que invalida una propiedad virtual también puede sellarse, que especifica que para las clases derivadas ya no es virtual. Por último, una propiedad puede declararse abstracta. Esto significa que no existe ninguna implementación en la clase, y las clases derivadas deben escribir su propia implementación. Para obtener más información sobre estas opciones, vea Clases y miembros de clase abstractos y sellados (Guía de programación de C#).

> **NOTE**
>
> Es un error usar un modificador virtual, abstract u override en un descriptor de acceso de una propiedad static.

Ejemplo

En este ejemplo se muestran las propiedades de solo lectura, estáticas y de instancia. Acepta el nombre del empleado desde el teclado, incrementa `NumberOfEmployees` en 1 y muestra el nombre del empleado y el número.

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's number:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
    Employee number: 108
    Employee name: Claude Vige
*/
```

Ejemplo

En este ejemplo se muestra cómo tener acceso a una propiedad en una clase base que está oculta mediante otra propiedad que tiene el mismo nombre en una clase derivada:

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    static void Main()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}", m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}", ((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

A continuación se muestran puntos importantes del ejemplo anterior:

- La propiedad `Name` de la clase derivada oculta la propiedad `Name` de la clase base. En dicho caso, el modificador `new` se usa en la declaración de la propiedad en la clase derivada:

  ```
  public new string Name
  ```

- La conversión `(Employee)` se usa para tener acceso a la propiedad oculta de la clase base:

  ```
  ((Employee)m1).Name = "Mary";
  ```

  Para obtener más información sobre cómo ocultar miembros, vea el Modificador new.

Ejemplo

En este ejemplo, dos clases, `Cube` y `Square`, implementan una clase abstracta, `Shape`, e invalidan su propiedad `Area` abstracta. Tenga en cuenta el uso del modificador override en las propiedades. El programa acepta el lado como una entrada y calcula las áreas del cuadrado y el cubo. También acepta el área como una entrada y calcula el

lado correspondiente para el cuadrado y el cubo.

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
```

```
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00

    Enter the area: 24
    Side of the square = 4.90
    Side of the cube = 2.00
*/
```

Vea también

- Guía de programación de C#
- Propiedades
- Propiedades de interfaz
- Propiedades autoimplementadas

Propiedades de interfaces (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

Las propiedades se pueden declarar en una interfaz. En el ejemplo siguiente se declara un descriptor de acceso de propiedad de interfaz:

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Normalmente, las propiedades de interfaz no tienen un cuerpo. Los descriptores de acceso indican si la propiedad es de lectura y escritura, de solo lectura o de solo escritura. A diferencia de las clases y las estructuras, la declaración de los descriptores de acceso sin cuerpo no declara una propiedad implementada automáticamente. A partir de C# 8.0, una interfaz puede definir una implementación predeterminada para los miembros, incluidas las propiedades. La definición de una implementación predeterminada para una propiedad en una interfaz es poco frecuente, ya que las interfaces no pueden definir campos de datos de instancia.

Ejemplo

En este ejemplo, la interfaz `IEmployee` tiene una propiedad de lectura y escritura, `Name`, y una propiedad de solo lectura, `Counter`. La clase `Employee` implementa la interfaz `IEmployee` y usa estas dos propiedades. El programa lee el nombre de un nuevo empleado y el número actual de empleados y muestra el nombre del empleado y el número de empleados calculado.

Puede usar el nombre completo de la propiedad, que hace referencia a la interfaz en la que se declara el miembro. Por ejemplo:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

En el ejemplo anterior se muestra la implementación de interfaz explícita. Por ejemplo, si la clase `Employee` implementa dos interfaces `ICitizen` y `IEmployee`, y ambas interfaces tienen la propiedad `Name`, la implementación del miembro de interfaz explícita será necesaria. Es decir, la siguiente declaración de propiedad:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implementa la propiedad `Name` en la interfaz `IEmployee`, mientras que la siguiente declaración:

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

implementa la propiedad `Name` en la interfaz `ICitizen`.

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string _name;
    public string Name  // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter  // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);
```

`210 Hazem Abolrous`

Resultados de ejemplo

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

Vea también

- Guía de programación de C#
- Propiedades
- Utilizar propiedades
- Comparación entre propiedades e indizadores
- Indizadores
- Interfaces

Restringir la accesibilidad del descriptor de acceso (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

Las partes get y set de una propiedad o un indexador se denominan *descriptores de acceso*. De forma predeterminada, estos descriptores de acceso tienen la misma visibilidad o nivel de acceso de la propiedad o el indexador al que pertenecen. Para obtener más información, vea Niveles de accesibilidad. En cambio, a veces resulta útil restringir el acceso a uno de estos descriptores de acceso. Normalmente, esto implica restringir la accesibilidad del descriptor de acceso `set`, mientras que se mantiene el descriptor de acceso `get` accesible públicamente. Por ejemplo:

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

En este ejemplo, una propiedad denominada `Name` define un descriptor de acceso `get` y `set`. El descriptor de acceso `get` recibe el nivel de accesibilidad de la propiedad, `public` en este caso, mientras que el descriptor de acceso `set` se restringe explícitamente al aplicar el modificador de acceso protected al propio descriptor de acceso.

Restricciones en los modificadores de acceso en descriptores de acceso

Usar los modificadores de descriptor de acceso en propiedades o indexadores está sujeto a estas condiciones:

- No puede usar los modificadores de descriptor de acceso en una interfaz o en una implementación explícita del miembro interface.
- Puede usar modificadores de descriptor de acceso solo si la propiedad o el indexador tienen los descriptores de acceso `set` y `get`. En este caso, se permite el modificador solo en uno de los dos descriptores de acceso.
- Si la propiedad o el indexador tienen un modificador override, el modificador de descriptor de acceso debe coincidir con el descriptor de acceso del descriptor de acceso invalidado, si lo hay.
- El nivel de accesibilidad del descriptor de acceso debe ser más restrictivo que el nivel de accesibilidad de la propiedad o el indexador.

Modificadores de acceso en descriptores de acceso de invalidación

Al invalidar una propiedad o un indexador, los descriptores de acceso invalidados deben ser accesibles para el código de invalidación. Además, la accesibilidad de la propiedad y el indexador y sus descriptores de acceso debe

coincidir con la propiedad y el indexador invalidados correspondientes y sus descriptores de acceso. Por ejemplo:

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

Implementar interfaces

Al usar un descriptor de acceso para implementar una interfaz, este no puede tener un modificador de acceso. En cambio, si implementa la interfaz con un descriptor de acceso, como `get`, el otro descriptor de acceso puede tener un modificador de acceso, como en el ejemplo siguiente:

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Dominio de accesibilidad de descriptor de acceso

Si usa un modificador de acceso en el descriptor de acceso, el dominio de accesibilidad del descriptor de acceso viene determinado por este modificador.

Si no ha usado un modificador de acceso en el descriptor de acceso, el dominio de accesibilidad del descriptor de

acceso viene determinado por el nivel de accesibilidad de la propiedad o el indexador.

Ejemplo

En el ejemplo siguiente, se incluyen tres clases: `BaseClass`, `DerivedClass` y `MainClass`. Hay dos propiedades en `BaseClass`, `Name` y `Id` en ambas clases. En el ejemplo, se muestra cómo la propiedad `Id` en `DerivedClass` se puede ocultar con la propiedad `Id` en `BaseClass` al usar un modificador de acceso restrictivo como protected o private. Por tanto, cuando asigna valores a esta propiedad, se llama en su lugar a la propiedad en la clase `BaseClass`. Si se reemplaza el modificador de acceso por public, la propiedad será accesible.

En el ejemplo, también se muestra que un modificador de acceso restrictivo, como `private` o `protected`, en el descriptor de acceso `set` de la propiedad `Name` en `DerivedClass` impide el acceso al descriptor de acceso y genera un error al asignárselo.

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}
```

```
}

class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Comentarios

Tenga en cuenta que, si reemplaza la declaración `new private string Id` por `new public string Id`, obtendrá el resultado:

```
Name and ID in the base class: Name-BaseClass, ID-BaseClass
```

```
Name and ID in the derived class: John, John123
```

Vea también

- Guía de programación de C#
- Propiedades
- Indizadores
- Modificadores de acceso

Procedimiento para declarar y usar propiedades de lectura y escritura (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Las propiedades proporcionan la comodidad de los miembros de datos públicos sin los riesgos que provienen del acceso sin comprobar, sin controlar y sin proteger a los datos de un objeto. Esto se consigue mediante los *descriptores de acceso*: métodos especiales que asignan y recuperan valores del miembro de datos subyacente. El descriptor de acceso set permite que los miembros de datos se asignen, y el descriptor de acceso get recupera los valores de los miembros de datos.

En este ejemplo se muestra una clase `Person` que tiene dos propiedades: `Name` (string) y `Age` (int). Ambas propiedades proporcionan descriptores de acceso `get` y `set`, de manera que se consideran propiedades de lectura y escritura.

Ejemplo

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }

        set
        {
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

class TestPerson
{
    static void Main()
    {
```

```
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/
```

Programación sólida

En el ejemplo anterior, las propiedades `Name` y `Age` son públicas e incluyen un descriptor de acceso `get` y `set`. Esto permite que cualquier objeto lea y escriba estas propiedades. En cambio, a veces esto es conveniente para excluir uno de los descriptores de acceso. Omitir el descriptor de acceso `set`, por ejemplo, hace que la propiedad sea de solo lectura:

```
public string Name
{
    get
    {
        return _name;
    }
}
```

De manera alternativa, puede exponer un descriptor de acceso públicamente pero hacer que el otro sea privado o esté protegido. Para obtener más información, vea Accesibilidad del descriptor de acceso asimétrico.

Una vez que se declaren las propiedades, pueden usarse como si fueran campos de la clase. Esto permite una sintaxis muy natural cuando ambos obtienen y establecen el valor de una propiedad, como se muestra en las instrucciones siguientes:

```
person.Name = "Joe";
person.Age = 99;
```

Tenga en cuenta que en un método `set` de la propiedad está disponible una variable `value` especial. Esta variable contiene el valor que el usuario ha especificado, por ejemplo:

```
_name = value;
```

Tenga en cuenta la sintaxis pura para incrementar la propiedad `Age` en un objeto `Person`:

```
person.Age += 1;
```

Si los métodos `set` y `get` independientes se han usado para modelar las propiedades, el código equivalente puede tener este aspecto:

```
person.SetAge(person.GetAge() + 1);
```

El método `ToString` se invalida en este ejemplo:

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Tenga en cuenta que `ToString` no se usa explícitamente en el programa. Se invoca de manera predeterminada mediante las llamadas a `WriteLine`.

Vea también

- Guía de programación de C#
- Propiedades
- Clases y structs

Propiedades autoimplementadas (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

En C# 3.0 y versiones posteriores, las propiedades implementadas automáticamente hacen que la declaración de propiedades sea más concisa cuando no es necesaria ninguna lógica adicional en los descriptores de acceso de la propiedad. También permite que el código de cliente cree objetos. Cuando se declara una propiedad tal como se muestra en el ejemplo siguiente, el compilador crea un campo de respaldo privado y anónimo al que solo se puede acceder con los descriptores de acceso de propiedad `get` y `set`.

Ejemplo

En el ejemplo siguiente se muestra una clase simple que tiene algunas propiedades implementadas automáticamente:

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Intialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

No se pueden declarar propiedades implementadas automáticamente en interfaces. Las propiedades implementadas automáticamente declaran un campo de respaldo de instancia privada y las interfaces no pueden declarar campos de instancia. Al declarar una propiedad en una interfaz sin definir un cuerpo, se declara una propiedad con descriptores de acceso que debe ser implementada por cada tipo que implemente esa interfaz.

En C# 6 y versiones posteriores, puede inicializar las propiedades implementadas automáticamente de forma similar a los campos:

```
public string FirstName { get; set; } = "Jane";
```

La clase que se muestra en el ejemplo anterior es mutable. El código de cliente puede cambiar los valores de los objetos una vez creados. En clases complejas que contienen el comportamiento importante (métodos) y los datos, suele ser necesario tener propiedades públicas. Pero para clases pequeñas o structs que simplemente encapsulan un conjunto de valores (datos) y tienen pocos comportamientos o ninguno, debe establecer que los objetos sean inmutables ya sea declarando el descriptor de acceso set como private (inmutable para los consumidores) o declarando solo un descriptor de acceso get (inmutable siempre excepto en el constructor). Para obtener más información, vea Procedimiento para implementar una clase ligera con propiedades autoimplementadas.

Vea también

- Propiedades
- Modificadores

Procedimiento para implementar una clase ligera con propiedades autoimplementadas (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

En este ejemplo se muestra cómo crear una clase ligera inmutable que solo sirve para encapsular un conjunto de propiedades autoimplementadas. Use este tipo de construcción en lugar de un struct cuando deba utilizar una semántica de tipo de referencia.

Puede crear una propiedad inmutable de dos maneras:

- Puede declarar el descriptor de acceso set como private. La propiedad solo se puede establecer dentro del tipo, pero es inmutable a los consumidores.

  Cuando se declara un descriptor de acceso `set` privado, no se puede usar un inicializador de objeto para inicializar la propiedad. Se debe utilizar un constructor o un método factory.

- Puede declarar solo el descriptor de acceso get, que hace que la propiedad sea inmutable en cualquier lugar excepto en el constructor del tipo.

En el ejemplo siguiente se muestra cómo una propiedad con solo el descriptor de acceso get es distinta de una con get y un conjunto privado.

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress
}
```

Ejemplo

En el siguiente ejemplo se muestran dos maneras de implementar una clase inmutable que tenga propiedades autoimplementadas. Cada forma declara una de las propiedades con un `set` privado y una de las propiedades solamente con un `get`. La primera clase usa un constructor solo para inicializar las propiedades y la segunda clase utiliza un método factory estático que llama a un constructor.

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
```

```
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = {"Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"};
        string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st Ave",
                              "12 108th St.", "89 E. 42nd St."};

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                         select Contact2.CreateContact(names[i], addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();
```

```
        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

El compilador crea campos de respaldo para cada propiedad autoimplementada. No se puede acceder a los campos directamente desde el código fuente.

Vea también

- Propiedades
- struct
- Inicializadores de objeto y colección

Métodos (Guía de programación de C#)

18/03/2020 • 19 minutes to read • Edit Online

Un método es un bloque de código que contiene una serie de instrucciones. Un programa hace que se ejecuten las instrucciones al llamar al método y especificando los argumentos de método necesarios. En C#, todas las instrucciones ejecutadas se realizan en el contexto de un método. El método `Main` es el punto de entrada para cada aplicación de C# y se llama mediante Common Language Runtime (CLR) cuando se inicia el programa.

> **NOTE**
>
> En este artículo se analizan los métodos denominados. Para obtener información sobre las funciones anónimas, vea Funciones anónimas (Guía de programación de C#).

Firmas de método

Los métodos se declaran en una clase, struct o interfaz especificando el nivel de acceso, como `public` o `private`, modificadores opcionales como `abstract` o `sealed`, el valor devuelto, el nombre del método y cualquier parámetro de método. Todas estas partes forman la firma del método.

> **NOTE**
>
> Un tipo de valor devuelto de un método no forma parte de la firma del método con el objetivo de sobrecargar el método. Sin embargo, forma parte de la firma del método al determinar la compatibilidad entre un delegado y el método que señala.

Los parámetros de método se encierran entre paréntesis y se separan por comas. Los paréntesis vacíos indican que el método no requiere parámetros. Esta clase contiene cuatro métodos:

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Acceso a métodos

Llamar a un método en un objeto es como acceder a un campo. Después del nombre del objeto, agregue un punto, el nombre del método y paréntesis. Los argumentos se enumeran entre paréntesis y están separados por comas. Los métodos de la clase `Motorcycle` se pueden llamar como en el ejemplo siguiente:

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

Parámetros de métodos frente a argumentos

La definición del método especifica los nombres y tipos de todos los parámetros necesarios. Si el código de llamada llama al métodos, proporciona valores concretos denominados argumentos para cada parámetro. Los argumentos deben ser compatibles con el tipo de parámetro, pero el nombre del argumento (si existe) utilizado en el código de llamada no tiene que ser el mismo que el parámetro con nombre definido en el método. Por ejemplo:

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaulates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Pasar por referencia frente a pasar por valor

De forma predeterminada, cuando se pasa una instancia de un tipo de valor a un método, se pasa su copia en lugar de la propia instancia. Por lo tanto, los cambios realizados en el argumento no tienen ningún efecto en la instancia original del método de llamada. Para pasar una instancia de tipo de valor por referencia, use la palabra clave `ref`. Para obtener más información, vea Pasar parámetros de tipo de valor (Guía de programación de C#).

Cuando se pasa un objeto de un tipo de referencia a un método, se pasa una referencia al objeto. Es decir, el método no recibe el objeto concreto, recibe un argumento que indica la ubicación del objeto. Si cambia un miembro del objeto mediante esta referencia, el cambio se refleja en el argumento del método de llamada, incluso si pasa el objeto por valor.

Crea un tipo de referencia mediante la palabra clave `class`, como se muestra en el siguiente ejemplo:

```
public class SampleRefType
{
    public int value;
}
```

Ahora, si se pasa un objeto basado en este tipo a un método, también se pasa una referencia al objeto. En el ejemplo siguiente se pasa un objeto de tipo `SampleRefType` al método `ModifyObject`:

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}
static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

Fundamentalmente, el ejemplo hace lo mismo que el ejemplo anterior en el que se pasa un argumento por valor a un método. Pero, debido a que se utiliza un tipo de referencia, el resultado es diferente. La modificación que se lleva a cabo en `ModifyObject` al campo `value` del parámetro, `obj`, también cambia el campo `value` del argumento, `rt`, en el método `TestRefType` . El método `TestRefType` muestra 33 como salida.

Para obtener más información sobre cómo pasar tipos de referencia por valor y por referencia, vea Pasar parámetros Reference-Type (Guía de programación de C#) y Tipos de referencia (Referencia de C#).

Valores devueltos

Los métodos pueden devolver un valor al autor de llamada. Si el tipo de valor devuelto, el tipo enumerado antes del nombre de método, no es `void`, el método puede devolver el valor mediante la utilización de la palabra clave `return` . Una instrucción con la palabra clave `return` seguida de un valor que coincide con el tipo de valor devuelto devolverá este valor al autor de llamada del método.

El valor puede devolverse al autor de la llamada mediante valor o, a partir de C# 7.0, mediante referencia. Los valores se devuelven al autor de la llamada mediante referencia si la palabra clave `ref` se usa en la firma del método y sigue cada palabra clave `return`. Por ejemplo, la siguiente firma del método y la instrucción return indican que el método devuelve nombres de variable `estDistance` mediante referencia al autor de la llamada.

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

La palabra clave `return` también detiene la ejecución del método. Si el tipo de valor devuelto es `void`, una instrucción `return` sin un valor también es útil para detener la ejecución del método. Sin la palabra clave

`return` , el método dejará de ejecutarse cuando alcance el final del bloque de código. Los métodos con un tipo de valor devuelto no nulo son necesarios para usar la palabra clave `return` para devolver un valor. Por ejemplo, estos dos métodos utilizan la palabra clave `return` para devolver enteros:

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

Para utilizar un valor devuelto de un método, el método de llamada puede usar la llamada de método en cualquier lugar; un valor del mismo tipo sería suficiente. También puede asignar el valor devuelto a una variable. Por ejemplo, los dos siguientes ejemplos de código logran el mismo objetivo:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Usar una variable local, en este caso, `result` , para almacenar un valor es opcional. La legibilidad del código puede ser útil, o puede ser necesaria si debe almacenar el valor original del argumento para todo el ámbito del método.

Para usar un valor devuelto mediante referencia de un método, debe declarar una variable local de tipo ref si pretende modificar su valor. Por ejemplo, si el método `Planet.GetEstimatedDistance` devuelve un valor Double mediante referencia, puede definirlo como una variable local de tipo ref con código como el siguiente:

```
ref int distance = plant
```

Devolver una matriz multidimensional de un método, `M` , que modifica el contenido de la matriz no es necesario si la función de llamada ha pasado la matriz a `M` . Puede devolver la matriz resultante de `M` para obtener un estilo correcto o un flujo funcional de valores, pero no es necesario porque C# pasa todos los tipos de referencia mediante valor, y el valor de una referencia de matriz es el puntero de la matriz. En el método `M` , los cambios en el contenido de la matriz los puede observar cualquier código que tenga una referencia a la matriz, como se muestra en el ejemplo siguiente:

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

Para obtener más información, consulta return.

Métodos asincrónicos

Mediante la característica asincrónica, puede invocar métodos asincrónicos sin usar definiciones de llamada explícitas ni dividir manualmente el código en varios métodos o expresiones lambda.

Si marca un método con el modificador async , puede usar el operador await en el método. Cuando el control alcanza una expresión await en el método asincrónico, el control se devuelve al autor de llamada y se progreso del método se suspende hasta que se completa la tarea esperada. Cuando se completa la tarea, la ejecución puede reanudarse en el método.

> **NOTE**
> Un método asincrónico vuelve al autor de llamada cuando encuentra el primer objeto esperado que aún no se ha completado o cuando llega al final del método asincrónico, lo que ocurra primero.

Un método asincrónico puede tener un tipo de valor devuelto de Task<TResult>, Task, o nulo. El tipo de valor devuelto nulo se utiliza principalmente para definir controladores de eventos, donde se requiere un tipo de valor devuelto nulo. No se puede esperar un método asincrónico que devuelve nulo, y el autor de llamada de un método con tipo de devolución nulo no puede capturar ninguna excepción producida por este.

En el ejemplo siguiente, `DelayAsync` es un método asincrónico con un tipo de valor devuelto de Task<TResult>. `DelayAsync` tiene una instrucción `return` que devuelve un entero. Por lo tanto, la declaración del método de `DelayAsync` debe tener un tipo de valor devuelto de `Task<int>`. Dado que el tipo de valor devuelto es `Task<int>`, la evaluación de la expresión `await` en `DoSomethingAsync` genera un entero, como se demuestra en la siguiente instrucción: `int result = await delayTask`.

El método `startButton_Click` es un ejemplo de un método asincrónico con un tipo de valor devuelto nulo. Dado que `DoSomethingAsync` es un método asincrónico, la tarea de la llamada a `DoSomethingAsync` debe esperar, como se muestra en la siguiente instrucción: `await DoSomethingAsync();`. El método `startButton_Click` debe definirse con el modificador `async` porque el método tiene una expresión `await` .

```
// using System.Diagnostics;
// using System.Threading.Tasks;

// This Click event is marked with the async modifier.
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    await DoSomethingAsync();
}

private async Task DoSomethingAsync()
{
    Task<int> delayTask = DelayAsync();
    int result = await delayTask;

    // The previous two statements may be combined into
    // the following statement.
    //int result = await DelayAsync();

    Debug.WriteLine("Result: " + result);
}

private async Task<int> DelayAsync()
{
    await Task.Delay(100);
    return 5;
}

// Output:
//  Result: 5
```

Un método asincrónico no puede declarar ningún parámetro ref u out, pero puede llamar a los métodos que tienen estos parámetros.

Para obtener más información sobre los métodos asincrónicos, vea Programación asincrónica con Async y Await, Flujo de control en programas asincrónicos y Tipos de valor devuelto de Async.

Definiciones de cuerpos de expresión

Es habitual tener definiciones de método que simplemente hacen las devoluciones de forma inmediata con el resultado de una expresión, o que tienen una sola instrucción como cuerpo del método. Hay un acceso directo de sintaxis para definir este método mediante `=>`:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

Si el método devuelve `void` o si es un método asincrónico, el cuerpo del método debe ser una expresión de instrucción (igual que con las expresiones lambda). Para las propiedades y los indexadores, solo deben leerse, y no usa la palabra clave de descriptor de acceso `get`.

Iterators

Un iterador realiza una iteración personalizada en una colección, como una lista o matriz. Un iterador usa la instrucción yield return para devolver cada elemento de uno en uno. Cuando se alcanza la instrucción yield return , se recuerda la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama el iterador.

Llame a un iterador a partir del código de cliente mediante una instrucción foreach .

El tipo de valor devuelto de un iterador puede ser IEnumerable, IEnumerable<T>, IEnumerator o IEnumerator<T>.

Para obtener más información, consulta Iteradores.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Clases y structs
- Modificadores de acceso
- Clases estáticas y sus miembros
- Herencia
- Clases y miembros de clase abstractos y sellados
- params
- return
- out
- ref
- Pasar parámetros

Funciones locales (Guía de programación de C#)

04/05/2020 • 19 minutes to read • Edit Online

A partir de C# 7.0, C# admite *funciones locales*. Las funciones locales son métodos privados de un tipo que están anidados en otro miembro. Solo se pueden llamar desde su miembro contenedor. Las funciones locales se pueden declarar en y llamar desde:

- Métodos, especialmente los métodos de iterador y asincrónicos
- Constructores
- Descriptores de acceso de propiedad
- Descriptores de acceso de un evento
- Métodos anónimos
- Expresiones lambda
- Finalizadores
- Otras funciones locales

En cambio, las funciones locales no se pueden declarar dentro de un miembro con forma de expresión.

> **NOTE**
> En algunos casos, puede usar una expresión lambda para implementar funcionalidad compatible también con una función local. Para ver una comparación, consulte Funciones locales frente a expresiones lambda.

Las funciones locales aclaran la intención del código. Cualquiera que lea el código puede ver que solo el método que lo contiene puede llamar al método. Para los proyectos de equipo, también hacen que sea imposible que otro desarrollador llame erróneamente al método de forma directa desde cualquier otro punto de la clase o estructura.

Sintaxis de función local

Una función local se define como un método anidado dentro de un miembro contenedor. Su definición tiene la siguiente sintaxis:

```
<modifiers: async | unsafe> <return-type> <method-name> <parameter-list>
```

Las funciones locales pueden usar los modificadores async y unsafe.

Tenga en cuenta que todas las variables locales que se definen en el miembro contenedor, incluidos sus parámetros de método, son accesibles en la función local.

A diferencia de una definición de método, una definición de función local no puede incluir el modificador de acceso de los miembros. Dado que todas las funciones locales son privadas, incluido un modificador de acceso, como la palabra clave `private`, se genera el error del compilador CS0106, "El modificador 'private' no es válido para este elemento".

> **NOTE**
> En las versiones anteriores a C# 8.0, las funciones locales no podían incluir el modificador `static`. Al incluir la palabra clave `static`, se genera el error del compilador CS0106, "El modificador 'static' no es válido para este elemento".

Además, los atributos no se pueden aplicar a la función local o a sus parámetros y parámetros de tipo.

En el ejemplo siguiente, se define una función local denominada `AppendPathSeparator` que es privada a un método denominado `GetText`:

```
using System;
using System.IO;

class Example
{
    static void Main()
    {
        string contents = GetText(@"C:\temp", "example.txt");
        Console.WriteLine("Contents of the file:\n" + contents);
    }

    private static string GetText(string path, string filename)
    {
         var sr = File.OpenText(AppendPathSeparator(path) + filename);
         var text = sr.ReadToEnd();
         return text;

         // Declare a local function.
         string AppendPathSeparator(string filepath)
         {
            if (! filepath.EndsWith(@"\"))
               filepath += @"\";

            return filepath;
         }
    }
}
```

Excepciones y funciones locales

Una de las características útiles de las funciones locales es que pueden permitir que las excepciones aparezcan inmediatamente. Para los iteradores de método, las excepciones aparecen solo cuando la secuencia devuelta se enumera y no cuando se recupera el iterador. Para los métodos asincrónicos, las excepciones producidas en un método asincrónico se observan cuando se espera la tarea devuelta.

En el ejemplo siguiente, se define un método `OddSequence` que enumera los números impares entre un intervalo especificado. Dado que pasa un número mayor de 100 al método de enumerador `OddSequence`, el método produce una excepción ArgumentOutOfRangeException. Como se muestra en el resultado del ejemplo, la excepción aparece solo cuando itera los números, y no al recuperar el enumerador.

```
using System;
using System.Collections.Generic;

class Example
{
   static void Main()
   {
      IEnumerable<int> ienum = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var i in ienum) //Line 11
      {
         Console.Write($"{i} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException("start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException("end must be less than or equal to 100."); //Line 22
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the following output:
//    Retrieved enumerator...
//
//    Unhandled Exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of
valid values.
//    Parameter name: end must be less than or equal to 100.
//       at Sequence.<GetNumericRange>d__1.MoveNext() in Program.cs:line 11
//       at Example.Main() in Program.cs:line 22
```

En su lugar, puede producir una excepción al realizar la validación y antes de recuperar el iterador al devolver el iterador de una función local, como se muestra en el ejemplo siguiente.

```
using System;
using System.Collections.Generic;

class Example
{
   static void Main()
   {
      IEnumerable<int> ienum = OddSequence(50, 110); //Line 8
      Console.WriteLine("Retrieved enumerator...");

      foreach (var i in ienum)
      {
         Console.Write($"{i} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException("start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException("end must be less than or equal to 100."); //Line 22
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      return GetOddSequenceEnumerator();

      IEnumerable<int> GetOddSequenceEnumerator()
      {
         for (int i = start; i <= end; i++)
         {
            if (i % 2 == 1)
               yield return i;
         }
      }
   }
}
// The example displays the following output:
//    Unhandled Exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of
valid values.
//    Parameter name: end must be less than or equal to 100.
//       at Sequence.<GetNumericRange>d__1.MoveNext() in Program.cs:line 8
//       at Example.Main() in Program.cs:line 22
```

Las funciones locales se pueden usar de forma similar, para controlar las excepciones fuera de la operación asincrónica. Normalmente, las excepciones producidas en el método asincrónico requieren que examine las excepciones internas de una excepción AggregateException. Las funciones locales permiten que el código genere un error inmediato y permiten que la excepción se produzca y observe de forma sincrónica.

En el ejemplo siguiente, se usa un método asincrónico denominado `GetMultipleAsync` para pausar durante un número especificado de segundos y se devuelve un valor que es un múltiplo aleatorio de ese número de segundos. El retraso máximo es de 5 segundos; se produce una excepción ArgumentOutOfRangeException si el valor es mayor que 5. Como se muestra en el ejemplo siguiente, la excepción que produce cuando se pasa un valor de 6 al método `GetMultipleAsync` se encapsula en una excepción AggregateException después de que el método `GetMultipleAsync` empiece a ejecutarse.

```
using System;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      int result = GetMultipleAsync(6).Result; //Line 8
      Console.WriteLine($"The returned value is {result:N0}");
   }

   static async Task<int> GetMultipleAsync(int secondsDelay)
   {
      Console.WriteLine("Executing GetMultipleAsync...");
      if (secondsDelay < 0 || secondsDelay > 5)
         throw new ArgumentOutOfRangeException("secondsDelay cannot exceed 5."); // Line 16

      await Task.Delay(secondsDelay * 1000);
      return secondsDelay * new Random().Next(2,10);
   }
}
// The example displays the following output:
//    Executing GetMultipleAsync...
//
//    Unhandled Exception: System.AggregateException:
//         One or more errors occurred. (Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.) --->
//         System.ArgumentOutOfRangeException: Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.
//       at Example.<GetMultiple>d__1.MoveNext() in Program.cs:line 16
//       --- End of inner exception stack trace ---
//       at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)
//       at System.Threading.Tasks.Task`1.GetResultCore(Boolean waitCompletionNotification)
//       at Example.Main() in C:\Users\ronpet\Documents\Visual Studio 2017\Projects\local-
functions\async1\Program.cs:line 8
```

Igual que hemos hecho con el iterador de método, podemos refactorizar el código de este ejemplo para realizar la validación antes de llamar al método asincrónico. Como se muestra en el resultado del ejemplo siguiente, la excepción ArgumentOutOfRangeException no se encapsula en una excepción AggregateException.

```
using System;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      int result = GetMultiple(6).Result; // Line 8
      Console.WriteLine($"The returned value is {result:N0}");
   }

   static Task<int> GetMultiple(int secondsDelay)
   {
      if (secondsDelay < 0 || secondsDelay > 5)
         throw new ArgumentOutOfRangeException("secondsDelay cannot exceed 5."); // Line 15

      return GetValueAsync();

      async Task<int> GetValueAsync()
      {
         Console.WriteLine("Executing GetValueAsync...");
         await Task.Delay(secondsDelay * 1000);
         return secondsDelay * new Random().Next(2,10);
      }
   }
}
// The example displays the following output:
//    Unhandled Exception: System.ArgumentOutOfRangeException:
//       Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.
//       at Example.GetMultiple(Int32 secondsDelay) in Program.cs:line 15
//       at Example.Main() in Program.cs:line 8
```

Funciones locales frente a expresiones lambda

A primera vista, las funciones locales y las expresiones lambda son muy similares. En muchos casos, la elección entre usar expresiones lambda y funciones locales es una cuestión de estilo y de preferencia personal, aunque hay diferencias que debe tener en cuenta acerca de dónde puede usar una u otra.

Vamos a examinar las diferencias entre las implementaciones de la función local y la expresión lambda del algoritmo factorial. Primero la versión que usa una función local:

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => (number < 2) ?
        1 : number * nthFactorial(number - 1);
}
```

Compare esa implementación con una versión que use expresiones lambda:

```
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = (number) => (number < 2) ?
        1 : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

Las funciones locales tienen nombres, mientras que las expresiones lambda son métodos anónimos que se asignan a variables que son tipos `Func` o `Action`. Cuando se declara una función local, los tipos de argumento y el tipo de valor devuelto forman parte de la declaración de función. En lugar de formar parte del cuerpo de la expresión lambda, los tipos de argumento y el tipo de valor devuelto forman parte de la declaración de tipo de variable de la expresión lambda. Estas dos diferencias pueden hacer que el código sea más claro.

Las funciones locales tienen reglas diferentes de las expresiones lambda para la asignación definitiva. Se puede hacer referencia a una declaración de función local desde cualquier ubicación del código que esté dentro del ámbito. Las expresiones lambda se deben asignar a una variable de delegado para poder acceder a ellas (o para poder llamarlas mediante el delegado que hace referencia a ellas). Observe que la versión con la expresión lambda debe declarar e inicializar la expresión lambda, `nthFactorial`, antes de definirla. De no hacerlo, se produce un error en tiempo de compilación por hacer referencia a `nthFactorial` antes de asignarlo. Estas diferencias implican que los algoritmos recursivos son más fáciles de crear usando funciones locales. Puede declarar y definir una función local que se llama a sí misma. Las expresiones lambda se deben declarar y se les debe asignar un valor predeterminado para que se les pueda volver a asignar un cuerpo que haga referencia a la misma expresión lambda.

Las reglas de asignación definitiva también afectan a las variables que se capturan con la función local o la expresión lambda. Tanto las funciones locales como las reglas de expresiones lambda exigen que las variables capturadas se asignen definitivamente en el momento en que la función local o la expresión lambda se convierta en un delegado. La diferencia está en que las expresiones lambda se convierten en delegados en el momento en que se declaran, mientras que las funciones locales se convierten en delegados solo cuando se usan como delegados. Si se declara una función local y solo se hace referencia a ella llamándola como un método, no se convertirá en un delegado. Esta regla le permite declarar una función local en cualquier ubicación adecuada de su ámbito de inclusión. Es habitual declarar funciones locales al final del método principal, después de cualquier instrucción "return".

En tercer lugar, el compilador puede efectuar un análisis estático que permita que las funciones locales asignen definitivamente variables capturadas en el ámbito de inclusión. Considere este ejemplo:

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

El compilador puede determinar que `LocalFunction` asigna `y` definitivamente cuando se le llama. Como se llama a `LocalFunction` antes de la instrucción `return`, `y` se asigna definitivamente en la instrucción `return`.

El análisis que proporciona el análisis de ejemplo posibilita la cuarta diferencia. Dependiendo de su uso, las funciones locales pueden evitar las asignaciones de montón que siempre son necesarias para las expresiones lambda. Si una función local no se convierte nunca en un delegado y ninguna de las variables capturadas con la función local se captura con otras expresiones lambda o funciones locales que se han convertido en delegados, el compilador puede evitar las asignaciones de montón.

Considere este ejemplo asincrónico:

```
public Task<string> PerformLongRunningWorkLambda(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    };

    return longRunningWorkImplementation();
}
```

La clausura de esta expresión lambda contiene las variables `address`, `index` y `name`. En el caso de las funciones locales, el objeto que implementa el cierre puede ser un tipo `struct`. Luego, ese tipo de estructura se pasaría por referencia a la función local. Esta diferencia de implementación supondría un ahorro en una asignación.

La creación de instancias necesaria para las expresiones lambda significa asignaciones de memoria adicionales, lo que puede ser un factor de rendimiento en rutas de acceso de código crítico en el tiempo. Las funciones locales no suponen esta sobrecarga. En el ejemplo anterior, la versión de las funciones locales tiene 2 asignaciones menos que la versión de la expresión lambda.

> **NOTE**
> La función local equivalente de este método también usa una clase para el cierre. Si el cierre de una función local se implementa como `class` o `struct` es un detalle de implementación. Una función local puede usar `struct` mientras una expresión lambda siempre usará `class`.

```
public Task<string> PerformLongRunningWork(string address, int index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required", paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index), message: "The index must be non-
negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name", paramName: nameof(name));

    return longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}
```

Una ventaja final que no se muestra en este ejemplo es que las funciones locales pueden implementarse como iteradores, con la sintaxis `yield return` para producir una secuencia de valores. La instrucción `yield return` no está permitida en las expresiones lambda.

Aunque las funciones locales pueden parecer redundantes con respecto a las expresiones lambda, en realidad

tienen finalidades y usos diferentes. Las funciones locales son más eficaces si se quiere escribir una función a la que solo se llame desde el contexto de otro método.

Vea también

- Métodos

Valores devueltos y variables locales de tipo ref

18/03/2020 • 13 minutes to read • Edit Online

A partir de C# 7.0, C# admite los valores devueltos de referencia (valores devueltos de tipo ref). Un valor devuelto de referencia permite que un método devuelva una referencia a una variable, en lugar de un valor, al autor de una llamada. El autor de la llamada puede tratar la variable devuelta como si se hubiera devuelto por valor o por referencia. El autor de la llamada puede crear una variable que sea una referencia al valor devuelto, lo que se conoce como una referencia local.

¿Qué es un valor devuelto de referencia?

La mayoría de los desarrolladores están familiarizados con pasar un argumento a un método llamado *por referencia*. Una lista de argumentos del método llamado incluye una variable que pasa la referencia. El autor de la llamada observa los cambios que ha realizado el método llamado a su valor. Un *valor devuelto de referencia* significa que un método devuelve una *referencia* (o un alias) a alguna variable. El ámbito de esa variable debe incluir el método. La duración de la variable debe extenderse más allá de la devolución del método. Las modificaciones en el valor del método devuelto por el autor de la llamada se realizan en la variable devuelta por el método.

Declarar que un método devuelve un *valor devuelto de referencia* indica que el método devuelve un alias a una variable. El diseño suele pretender que el código de llamada acceda a dicha variable con el alias, incluso para modificarla. Por tanto, los métodos devueltos por referencia no pueden tener el tipo de valor devuelto `void`.

Hay algunas restricciones en la expresión que un método puede devolver como un valor devuelto de referencia. Entre las restricciones se incluyen:

- El valor devuelto debe tener una duración que se extienda más allá de la ejecución del método. En otras palabras, no puede tratarse de una variable local del método que la devuelve. Puede ser una instancia o un campo estático de una clase, o puede ser un argumento pasado al método. Al intentar devolver una variable local, se genera el error del compilador CS8168, "No se puede devolver por referencia la variable local 'obj' porque no es de tipo ref".

- El valor devuelto no puede ser el literal `null`. Si devuelve `null`, se genera el error del compilador CS8156, "No se puede usar una expresión en este contexto porque no se puede devolver por referencia".

  Un método con un valor devuelto de referencia puede devolver un alias a una variable cuyo valor es actualmente el valor NULL (sin instancias) o un tipo de valor que admite un valor NULL para un tipo de valor.

- El valor devuelto no puede ser una constante, un miembro de enumeración, el valor devuelto por valor desde una propiedad o un método de `class` o `struct`. Si infringe esta regla, se genera el error del compilador CS8156, "No se puede usar una expresión en este contexto porque no se puede devolver por referencia".

Además, los valores devueltos por referencia no se permiten en métodos asincrónicos. Un método asincrónico puede volver antes de que haya terminado de ejecutarse, mientras que su valor devuelto aún no se conoce.

Definir un valor devuelto de tipo ref

Un método que devuelve un *valor devuelto de referencia* debe cumplir las dos condiciones siguientes:

- La firma del método incluye la palabra clave ref delante del tipo de valor devuelto.

- Cada instrucción return del cuerpo del método incluye la palabra clave ref delante del nombre de la instancia devuelta.

En el método siguiente se muestra un método que cumple estas condiciones y devuelve una referencia a un objeto `Person` denominado `p`:

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Usar un valor devuelto de tipo ref

El valor devuelto de tipo ref es un alias para otra variable en el ámbito del método llamado. Puede interpretar cualquier uso del valor devuelto tipo ref como si se usara la variable a la que se asigna el alias:

- Al asignar su valor, se asigna un valor a la variable a la que se asigna el alias.
- Al leer su valor, se lee un valor a la variable a la que se asigna el alias.
- Si la devolución se realiza *por referencia*, entonces devuelve un alias a esa misma variable.
- Si pasa el valor a otro método *por referencia*, pasará una referencia a la variable a la que se asigna el alias.
- Al asignar un alias local tipo ref, crea un alias para la misma variable.

Variables locales de tipo ref

Suponga que el método `GetContactInformation` se declara como un valor devuelto tipo ref:

```
public ref Person GetContactInformation(string fname, string lname)
```

Una asignación por valor lee el valor de una variable y lo asigna a una nueva variable:

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

La asignación anterior declara `p` como una variable local. Su valor inicial se copia a partir de la lectura del valor devuelto por `GetContactInformation`. Las asignaciones futuras a `p` no cambiarán el valor de la variable devuelta por `GetContactInformation`. La variable `p` ya no es un alias de la variable devuelta.

Se declara una variable *local tipo ref* para copiar el alias en el valor original. En la siguiente asignación, `p` es un alias para la variable devuelta desde `GetContactInformation`.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

El uso posterior de `p` es lo mismo que usar la variable devuelta por `GetContactInformation`, porque `p` es un alias para dicha variable. Los cambios realizados en `p` también modifican la variable devuelta desde `GetContactInformation`.

La palabra clave `ref` se usa antes de la declaración de variable local *y* antes de la llamada al método.

Puede acceder a un valor por referencia de la misma manera. En algunos casos, acceder a un valor por referencia aumenta el rendimiento, ya que evita una operación de copia potencialmente cara. Por ejemplo, en la instrucción siguiente se muestra cómo es posible definir un valor local de referencia que se usa para hacer referencia a un valor.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

La palabra clave `ref` se usa antes de la declaración de variable local *y* antes del valor en el segundo ejemplo. Si no se incluyen ambas palabras clave `ref` en la asignación y declaración de variable en ambos ejemplos, se produce el error del compilador CS8172, "No se puede inicializar una variable por referencia con un valor".

Antes de C# 7.3, las variables locales de tipo ref no se podían reasignar para hacer referencia a otro almacenamiento después de haberse inicializado. Esta restricción se ha quitado. En el ejemplo siguiente, se muestra una reasignación:

```
ref VeryLargeStruct reflocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Las variables locales de tipo ref todavía deben inicializarse cuando se declaran.

Valores devueltos y variables locales de tipo ref: un ejemplo

En el ejemplo siguiente, se define una clase `NumberStore` que almacena una matriz de valores enteros. El método `FindNumber` devuelve por referencia el primer número que es mayor o igual que el número que se pasa como argumento. Si ningún número es mayor o igual que el argumento, el método devuelve el número en el índice 0.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

En el ejemplo siguiente, se llama al método `NumberStore.FindNumber` para recuperar el primer valor que es mayor o igual que 16. Después, el autor de la llamada duplica el valor devuelto por el método. En el resultado del ejemplo se muestra el cambio reflejado en el valor de los elementos de matriz de la instancia `NumberStore`.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//       Original sequence: 1 3 7 15 31 63 127 255 511 1023
//       New sequence:      1 3 7 15 62 63 127 255 511 1023
```

Sin que se admitan los valores devueltos de referencia, este tipo de operación se realiza al devolver el índice del elemento de matriz junto con su valor. Después, el autor de la llamada puede usar este índice para modificar el valor en una llamada al método independiente. En cambio, el autor de la llamada también puede modificar el

índice para tener acceso a otros valores de matriz y, posiblemente, modificarlos.

En el siguiente ejemplo, se muestra cómo el método `FindNumber` podría modificarse después de C# 7.3 para usar la reasignación local de tipo ref:

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && numbers[ctr] >= target)
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

Esta segunda versión es más eficaz con secuencias mayores en escenarios donde el número que se busca está más cerca del final de la matriz.

Vea también

- ref (palabra clave)
- Escritura de código seguro y eficaz

Pasar parámetros (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

En C#, los argumentos se pueden pasar a parámetros por valor o por referencia. El paso de parámetros por referencia permite a los miembros de funciones, métodos, propiedades, indexadores, operadores y constructores cambiar el valor de los parámetros y hacer que ese cambio persista en el entorno de la llamada. Para pasar un parámetro por referencia con la intención de cambiar el valor, use la palabra clave `ref` o `out`. Para pasar un parámetro por referencia con la intención de evitar la copia pero no modificar el valor, use el modificador `in`. En los ejemplos de este tema, para simplificar, solo se usa la palabra clave `ref`. Para obtener más información sobre la diferencia entre `in`, `ref` y `out`, vea in, ref y out.

En el ejemplo siguiente se muestra la diferencia entre los parámetros de valor y de referencia.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

Para obtener más información, consulta los temas siguientes:

- Pasar parámetros de tipo de valor
- Pasar parámetros Reference-Type

Especificación del lenguaje C#

Para obtener más información, vea la sección Listas de argumentos de Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Métodos

Pasar parámetros de tipo de valor (Guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

Una variable tipo de valor contiene sus datos directamente, en oposición a la variable tipo de referencia, que contiene una referencia a sus datos. Pasar una variable tipo de valor a un método en función del valor significa pasar una copia de la variable al método. Ningún cambio realizado en el parámetro dentro del método afecta a los datos originales almacenados en la variable de argumentos. Si desea que el método al que se llama cambie el valor del argumento, debe pasarlo en función de la referencia, con la palabra clave ref o out. También puede usar la palabra clave in para pasar un parámetro de valor por referencia para evitar la copia, a la vez que se asegura de que el valor no se modificará. Para simplificar, en el ejemplo siguiente se usa `ref`.

Pasar tipos de valor en función del valor

En el ejemplo siguiente se muestra cómo pasar los parámetros de tipo de valor en función del valor. La variable `n` se pasa en función del valor al método `SquareIt`. Los cambios que tienen lugar dentro del método no tienen ningún efecto en el valor original de la variable.

```
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n);  // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

La variable `n` es un tipo de valor. Contiene sus datos, el valor `5`. Cuando se invoca `SquareIt`, el contenido de `n` se copia en el parámetro `x`, que se multiplica dentro del método. En `Main`, sin embargo, el valor de `n` es el mismo después de llamar al método `SquareIt` que el que era antes. El cambio que tiene lugar dentro del método solo afecta a la variable local `x`.

Pasar tipos de valor en función de la referencia

El ejemplo siguiente es el mismo que el anterior, salvo que el argumento se pasa como un parámetro `ref`. El valor del argumento subyacente, `n`, se cambia cuando se modifica `x` en el método.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n);  // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

En este ejemplo, no es el valor de `n` el que se pasa, sino una referencia a `n`. El parámetro `x` no es un entero; se trata de una referencia a `int`, en este caso, una referencia a `n`. Por tanto, cuando `x` se multiplica dentro del método, lo que realmente se multiplica es a lo que `x` hace referencia, `n`.

Intercambiar tipos de valor

Un ejemplo común de modificación de valores de argumentos es un método de intercambio, donde se pasan dos variables al método, y el método intercambia su contenido. Debe pasar los argumentos al método de intercambio en función de la referencia. De lo contrario, se intercambian copias locales de los parámetros dentro del método, y no se produce ningún cambio en el método de llamada. En el ejemplo siguiente se intercambian valores enteros.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Al llamar al método `SwapByRef`, use la palabra clave `ref` en la llamada, como se muestra en el ejemplo siguiente.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
/* Output:
    i = 2  j = 3
    i = 3  j = 2
*/
```

Vea también

- Guía de programación de C#
- Pasar parámetros
- Pasar parámetros Reference-Type

Pasar parámetros Reference-Type (Guía de programación de C#)

18/03/2020 • 6 minutes to read • Edit Online

Una variable de un tipo de referencia no contiene sus datos directamente, contiene una referencia a sus datos. Al pasar un parámetro de tipo de referencia por valor, es posible cambiar los datos que pertenecen al objeto al que se hace referencia, como el valor de un miembro de clase. En cambio, no se puede cambiar el valor de la propia referencia; por ejemplo, no puede usar la misma referencia para asignar memoria para un nuevo objeto y hacer que persista fuera del método. Para ello, pase el parámetro mediante las palabras clave ref u out. Para simplificar, en el ejemplo siguiente se usa `ref`.

Pasar tipos de referencia en función del valor

En el ejemplo siguiente, se muestra cómo pasar un parámetro de tipo de referencia, `arr`, en función del valor a un método, `Change`. Dado que el parámetro es una referencia a `arr`, es posible cambiar los valores de los elementos de matriz. En cambio, el intento de volver a asignar el parámetro a otra ubicación de memoria solo funciona dentro del método y no afecta a la variable original, `arr`.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888;  // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4};   // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr
[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr
[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

En el ejemplo anterior, la matriz, `arr`, que es un tipo de referencia, se pasa al método sin el parámetro `ref`. En tal caso, se pasa al método una copia de la referencia, que apunta a `arr`. El resultado muestra que es posible que el método cambie el contenido de un elemento de matriz, en este caso de `1` a `888`. En cambio, si se asigna una nueva porción de memoria al usar el operador new dentro del método `Change`, la variable `pArray` hace referencia a una nueva matriz. Por tanto, cualquier cambio que hubiese después no afectará a la matriz original, `arr`, que se ha creado dentro de `Main`. De hecho, se crean dos matrices en este ejemplo, una dentro de `Main` y otra dentro del método `Change`.

Pasar tipos de referencia en función de la referencia

El ejemplo siguiente es el mismo que el anterior, salvo que la palabra clave `ref` se agrega a la llamada y al encabezado de método. Los cambios que tengan lugar en el método afectan a la variable original en el programa que realiza la llamada.

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

Todos los cambios que tienen lugar dentro del método afectan a la matriz original en `Main`. De hecho, la matriz original se reasigna mediante el operador `new`. Por tanto, después de llamar al método `Change`, cualquier referencia a `arr` apunta a la matriz de cinco elementos, que se crea en el método `Change`.

Intercambiar dos cadenas

Intercambiar cadenas es un buen ejemplo de pasar parámetros de tipo de referencia en función de la referencia. En el ejemplo, se inicializan dos cadenas, `str1` y `str2`, en `Main` y se pasan al método `SwapStrings` como parámetros modificados por la palabra clave `ref`. Las dos cadenas se intercambian dentro del método y también dentro de `Main`.

```
class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
    // The string parameter is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);   // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
    Inside Main, before swapping: John Smith
    Inside the method: Smith John
    Inside Main, after swapping: Smith John
*/
```

En este ejemplo, los parámetros tienen que pasarse en función de la referencia para afectar a las variables en el programa que realiza la llamada. Si quita la palabra clave `ref` tanto del encabezado de método como de la llamada al método, no se llevará a cabo ningún cambio en el programa que realiza la llamada.

Para obtener más información sobre las cadenas, vea string.

Vea también

- Guía de programación de C#
- Pasar parámetros
- ref
- in
- out
- Tipos de referencia

Procedimiento para saber las diferencias entre pasar a un método un struct y una referencia a clase (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

En el ejemplo siguiente se muestra cómo pasar un struct a un método es diferente de pasar una instancia de clase a un método. En el ejemplo, los dos argumentos (struct e instancia de clase) se pasan mediante valor, y ambos métodos cambian el valor de un campo del argumento. En cambio, los resultados de los dos métodos no son los mismos porque lo que se pasa cuando pasa un struct difiere de lo que se pasa cuando pasa una instancia de una clase.

Como un struct es un tipo de valor, cuando pasa un struct mediante valor a un método, el método recibe y funciona en una copia del argumento struct. El método no tiene acceso al struct original en el método de llamada y, por lo tanto, no puede cambiarlo de ninguna manera. El método solo puede cambiar la copia.

Una instancia de clase es un tipo de referencia, no un tipo de valor. Cuando un tipo de referencia se pasa mediante valor a un método, el método recibe una copia de la referencia a la instancia de clase. Es decir, el método al que se llama recibe una copia de la dirección de la instancia y el método de llamada conserva la dirección original de la instancia. La instancia de clase en el método de llamada tiene una dirección, el parámetro en el método que se ha llamado tiene una copia de la dirección, y ambas direcciones hacen referencia al mismo objeto. Como el parámetro solo contiene una copia de la dirección, el método al que se ha llamado no puede cambiar la dirección de la instancia de clase en el método de llamada. En cambio, el método al que se ha llamado puede usar la copia de la dirección para acceder a los miembros de clase a los que hacen referencia la dirección original y la copia de la dirección. Si el método al que se ha llamado cambia un miembro de clase, la instancia de clase original en el método de llamada también cambia.

En el resultado del ejemplo siguiente se ilustra la diferencia. El valor del campo `willIChange` de la instancia de clase se cambia mediante la llamada al método `ClassTaker` porque el método usa la dirección en el parámetro para buscar el campo especificado de la instancia de clase. El campo `willIChange` del struct en el método de llamada no se cambia mediante la llamada al método `StructTaker` porque el valor del argumento es una copia del propio struct, no una copia de su dirección. `StructTaker` cambia la copia, y la copia se pierde cuando se completa la llamada a `StructTaker`.

Ejemplo

```
class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

Vea también

- Guía de programación de C#
- Clases
- Tipos de estructura
- Pasar parámetros

Variables locales con asignación implícita de tipos (Guía de programación de C#)

29/06/2020 • 9 minutes to read • Edit Online

Las variables locales pueden declararse sin proporcionar un tipo explícito. La palabra clave `var` indica al compilador que infiera el tipo de la variable a partir de la expresión de la derecha de la instrucción de inicialización. El tipo inferido puede ser un tipo integrado, un tipo anónimo, un tipo definido por el usuario o un tipo definido en la biblioteca de clases .NET. Para obtener más información sobre cómo inicializar las matrices con `var`, vea Matrices con tipo implícito.

Los ejemplos siguientes muestran distintas formas en que se pueden declarar variables locales con `var`:

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

Es importante comprender que la palabra clave `var` no significa "variant" ni indica que la variable esté débilmente tipada o esté enlazada en tiempo de ejecución. Solo significa que el compilador determina y asigna el tipo más adecuado.

La palabra clave `var` se puede usar en los contextos siguientes:

- En variables locales (variables declaradas en el ámbito del método), tal y como se muestra en el ejemplo anterior.
- En una instrucción de inicialización for.

  ```
  for (var x = 1; x < 10; x++)
  ```

- En una instrucción de inicialización foreach.

  ```
  foreach (var item in list) {...}
  ```

- En una instrucción using.

```
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

Para obtener más información, vea Procedimiento para usar matrices y variables locales con tipo implícito en expresiones de consulta.

var y tipos anónimos

En muchos casos, el uso de `var` es opcional y es simplemente una comodidad sintáctica. Pero cuando una variable se inicializa con un tipo anónimo, la variable se debe declarar como `var` si necesita acceder más adelante a las propiedades del objeto. Se trata de un escenario común en expresiones de consulta LINQ. Para obtener más información, consulte Tipos anónimos (Guía de programación de C#).

Desde el punto de vista del código fuente, un tipo anónimo no tiene nombre. Por lo tanto, si una variable de consulta se ha inicializado con `var`, la única manera de tener acceso a las propiedades de la secuencia de objetos devuelta consiste en usar `var` como el tipo de la variable de iteración en la instrucción `foreach`.

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper, ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Comentarios

Las siguientes restricciones se aplican a las declaraciones de variable con tipo implícito:

- `var` solo se puede usar cuando una variable local se declara e inicializa en la misma instrucción; la variable no se puede inicializar en null ni en un grupo de métodos o una función anónima.
- `var` no se puede usar en campos en el ámbito de clase.
- Las variables declaradas con `var` no se pueden usar en la expresión de inicialización. En otras palabras, esta expresión es válida: `int i = (i = 20);`, pero esta expresión genera un error en tiempo de compilación: `var i = (i = 20);`
- No se pueden inicializar varias variables con tipo implícito en la misma instrucción.
- Si en el ámbito se encuentra un tipo con nombre `var`, la palabra clave `var` se resolverá en ese nombre de tipo y no se tratará como parte de una declaración de variable local con tipo implícito.

El establecimiento de tipos implícitos con la palabra clave `var` solo puede aplicarse a variables en el ámbito del método local. El establecimiento de tipos implícitos no está disponible para los campos de clase ya que el compilador C# encontraría una paradoja lógica al procesar el código: el compilador necesita conocer el tipo de campo, pero no puede determinar el tipo hasta que se analiza la expresión de asignación, y no se puede evaluar la expresión sin conocer el tipo. Observe el código siguiente:

```
private var bookTitles;
```

`bookTitles` es un campo de clase dado el tipo `var`. Como el campo no tiene ninguna expresión que evaluar, es imposible que el compilador pueda inferir el tipo `bookTitles` que se supone que es. Además, agregar una expresión al campo (como se haría con una variable local) también es suficiente:

```
private var bookTitles = new List<string>();
```

Cuando el compilador encuentra campos durante la compilación de código, registra el tipo de cada campo antes de procesar las expresiones asociadas a él. El compilador encuentra la misma paradoja intentando analizar `bookTitles`: necesita saber el tipo de campo, pero el compilador normalmente determinaría el tipo de `var` analizando la expresión, lo que no es posible sin conocer de antemano el tipo.

Es posible que `var` también pueda resultar útil con expresiones de consulta en las que es difícil determinar el tipo construido exacto de la variable de consulta. Esto puede ocurrir con las operaciones de agrupamiento y ordenación.

La palabra clave `var` también puede ser útil cuando resulte tedioso escribir el tipo específico de la variable en el teclado, o sea obvio o no aumente la legibilidad del código. Un ejemplo en el que `var` resulta útil de esta manera es cuando se usa con tipos genéricos anidados, como los que se emplean con las operaciones de grupo. En la siguiente consulta, el tipo de la variable de consulta es `IEnumerable<IGrouping<string, Student>>`. Siempre que usted y otras personas que deban mantener el código comprendan esto, no hay ningún problema con el uso de tipos implícitos por comodidad y brevedad.

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

El uso de `var` ayuda a simplificar el código, pero debe quedar restringido a los casos en los que sea necesario, o cuando haga que el código sea más fácil de leer. Para obtener más información sobre cuándo usar `var` correctamente, vea la sección Variables locales con asignación implícita de tipos en el artículo sobre directrices de codificación de C#.

Vea también

- Referencia de C#
- Matrices con tipo implícito
- Procedimiento para usar matrices y variables locales con tipo implícito en expresiones de consulta
- Tipos anónimos
- Inicializadores de objeto y colección
- var
- LINQ en C#

- LINQ (Language Integrated Query)
- for
- foreach, in
- using (instrucción)

Procedimiento para usar matrices y variables locales con tipo implícito en expresiones de consulta (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

Puede usar variables locales con tipo implícito siempre que quiera que el compilador determine el tipo de una variable local. Debe usar variables locales con tipo implícito para almacenar tipos anónimos, que a menudo se usan en las expresiones de consulta. En los ejemplos siguientes, se muestran los usos obligatorios y opcionales de las variables locales con tipo implícito en las consultas.

Las variables locales con tipo implícito se declaran mediante la palabra clave contextual var. Para obtener más información, vea Variables locales con asignación implícita de tipos y Matrices con asignación implícita de tipos.

Ejemplo

En el ejemplo siguiente, se muestra un escenario común en el que la palabra clave `var` es necesaria: una expresión de consulta que genera una secuencia de tipos anónimos. En este escenario, la variable de consulta y la variable de iteración en la instrucción `foreach` deben escribirse de forma implícita mediante el uso de `var` porque no se tiene acceso a un nombre de tipo para el tipo anónimo. Para obtener más información sobre los tipos anónimos, vea Tipos anónimos.

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName, anonType.LastName);
    }
}
```

Ejemplo

En el ejemplo siguiente, se usa la palabra clave `var` en una situación similar, pero en la que el uso de `var` es opcional. Dado que `student.LastName` es una cadena, la ejecución de la consulta devuelve una secuencia de cadenas. Por tanto, el tipo de `queryID` podría declararse como `System.Collections.Generic.IEnumerable<string>` en lugar de `var`. La palabra clave `var` se usa por comodidad. En el ejemplo, la variable de iteración en la instrucción `foreach` se escribe de forma explícita como una cadena, pero se podría declarar mediante `var`. Dado que el tipo de la variable de iteración no es un tipo anónimo, el uso de `var` es opcional, no es obligatorio. Recuerde que `var` no es un tipo, sino una instrucción para que el compilador deduzca y asigne el tipo.

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

Vea también

- Guía de programación de C#
- Métodos de extensión
- LINQ (Language Integrated Query)
- var
- LINQ en C#

Métodos de extensión (Guía de programación de C#)

29/06/2020 • 17 minutes to read • Edit Online

Los métodos de extensión permiten "agregar" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original. Los métodos de extensión son métodos estáticos, pero se les llama como si fueran métodos de instancia en el tipo extendido. En el caso del código de cliente escrito en C#, F# y Visual Basic, no existe ninguna diferencia aparente entre llamar a un método de extensión y llamar a los métodos definidos en un tipo.

Los métodos de extensión más comunes son los operadores de consulta LINQ estándar, que agregan funciones de consulta a los tipos System.Collections.IEnumerable y System.Collections.Generic.IEnumerable<T> existentes. Para usar los operadores de consulta estándar, inclúyalos primero en el ámbito con una directiva `using System.Linq`. A partir de ese momento, cualquier tipo que implemente IEnumerable<T> parecerá tener métodos de instancia como GroupBy, OrderBy, Average, etc. Puede ver estos métodos adicionales en la finalización de instrucciones de IntelliSense al escribir "punto" después de una instancia de un tipo IEnumerable<T>, como List<T> o Array.

Ejemplo de OrderBy

En el ejemplo siguiente se muestra cómo llamar al método `OrderBy` de operador de consulta estándar en una matriz de enteros. La expresión entre paréntesis es una expresión lambda. Muchos operadores de consulta estándar toman expresiones lambda como parámetros, pero no es un requisito para los métodos de extensión. Para obtener más información, vea Expresiones lambda.

```
class ExtensionMethods2
{

    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
//Output: 10 15 21 26 39 45
```

Los métodos de extensión se definen como métodos estáticos, pero se les llama usando la sintaxis de método de instancia. Su primer parámetro especifica en qué tipo funciona el método. El parámetro va precedido del modificador this. Los métodos de extensión únicamente se encuentran dentro del ámbito cuando el espacio de nombres se importa explícitamente en el código fuente con una directiva `using`.

En el ejemplo siguiente se muestra un método de extensión definido para la clase System.String. Se define dentro de una clase estática no anidada y no genérica:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                             StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

El método de extensión `WordCount` se puede incluir en el ámbito con esta directiva `using`:

```
using ExtensionMethods;
```

También se le puede llamar desde una aplicación con esta sintaxis:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

El método de extensión se invoca en el código con la sintaxis de método de instancia. El lenguaje intermedio (IL) generado por el compilador convierte el código en una llamada en el método estático. El principio de encapsulación no se infringe realmente. Los métodos de extensión no pueden tener acceso a las variables privadas en el tipo que extienden.

Para obtener más información, vea Procedimiento para implementar e invocar un método de extensión personalizado.

En general, probablemente se llamará a métodos de extensión con mucha más frecuencia de la que se implementarán métodos propios. Dado que los métodos de extensión se llaman con la sintaxis de método de instancia, no se requieren conocimientos especiales para usarlos desde el código de cliente. Para habilitar los métodos de extensión para un tipo determinado, basta con agregar una directiva `using` para el espacio de nombres en el que se definen los métodos. Por ejemplo, para usar los operadores de consulta estándar, agregue esta directiva `using` al código:

```
using System.Linq;
```

(Puede que haya que agregar también una referencia a System.Core.dll). Observará que los operadores de consulta estándar aparecen ahora en IntelliSense como métodos adicionales disponibles para la mayoría de los tipos IEnumerable<T>.

Enlazar métodos de extensión en tiempo de compilación

Se pueden usar métodos de extensión para ampliar una clase o interfaz, pero no para invalidarlas. Nunca se llamará a un método de extensión con el mismo nombre y signatura que un método de interfaz o clase. En tiempo de compilación, los métodos de extensión siempre tienen menos prioridad que los métodos de instancia definidos en el propio tipo. En otras palabras, si un tipo tiene un método denominado `Process(int i)` y hay un método de extensión con la misma signatura, el compilador siempre se enlazará al método de instancia. Cuando el compilador encuentra una invocación de método, primero busca una coincidencia en los métodos de instancia del tipo. Si no la hay, buscará cualquier método de extensión definido para el tipo y se enlazará al primer método de extensión que encuentre. En el ejemplo siguiente se muestra cómo determina el compilador a qué método de extensión o de instancia enlazarse.

Ejemplo

En el ejemplo siguiente se muestran las reglas que el compilador de C# sigue para determinar si se va a enlazar una llamada a método a un método de instancia del tipo o a un método de extensión. La clase estática `Extensions` contiene métodos de extensión definidos para cualquier tipo que implemente `IMyInterface`. Las clases `A`, `B` y `C` implementan la interfaz.

Nunca se llama al método de extensión `MethodB`, porque su nombre y signatura coinciden exactamente con los métodos ya implementados por las clases.

Si el compilador no encuentra un método de instancia con una signatura coincidente, se enlazará a un método de extensión coincidente, si existe.

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}

// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;
```

```
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // A has a method that matches the signature of the following call
            // to MethodB.
            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);           // B.MethodA(int)
            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but
            // class Extension does.
            b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // C contains an instance method that matches each of the following
            // method calls.
            c.MethodA(1);           // C.MethodA(object)
            c.MethodA("hello");     // C.MethodA(object)
            c.MethodB();            // C.MethodB()
        }
    }
}
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
```

```
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

Patrones de uso común

Funcionalidad de colección

En el pasado, era habitual crear "Clases de colección" que implementaban la interfaz System.Collections.Generic.IEnumerable<T> para un tipo especificado e incluían una funcionalidad que actuaba en colecciones de ese tipo. Aunque no hay ningún problema con la creación de este tipo de objeto de colección, se puede lograr la misma funcionalidad mediante una extensión en System.Collections.Generic.IEnumerable<T>. Las extensiones tienen la ventaja de permitir que se llame a la funcionalidad desde cualquier colección como System.Array o System.Collections.Generic.List<T> que implementa System.Collections.Generic.IEnumerable<T> en ese tipo. Encontrará un ejemplo de esto mediante una matriz de Int32 anteriormente en este artículo.

Funcionalidad específica de la capa

Al usar Onion Architecture (Arquitectura cebolla) u otro diseño de la aplicación por niveles, es habitual tener un conjunto de entidades de dominio u objetos de transferencia de datos que se pueden usar para la comunicación entre los límites de la aplicación. Por regla general, estos objetos no contienen ninguna funcionalidad o contienen únicamente una funcionalidad mínima que se aplica a todas las capas de la aplicación. Los métodos de extensión se pueden usar para agregar una funcionalidad específica de cada capa de la aplicación sin cargar el objeto con métodos no necesarios o deseados en otras capas.

```
public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

Ampliación de tipos predefinidos

En lugar de crear objetos cuando es necesario crear una funcionalidad reutilizable, a menudo podemos ampliar un tipo existente como un tipo CLR o de .NET. Como ejemplo, si no usamos métodos de extensión, podemos crear una clase `Engine` o `Query` para ejecutar una consulta en un servidor SQL Server al que se puede llamar desde varias ubicaciones en nuestro código. Sin embargo, en su lugar, podemos ampliar la clase System.Data.SqlClient.SqlConnection mediante métodos de extensión para realizar esa consulta desde cualquier lugar en el que tengamos una conexión a un servidor SQL Server. Otros ejemplos podrían ser la adición de una funcionalidad común a la clase System.String, la ampliación de las funcionalidades de procesamiento de datos de los objetos System.IO.File y System.IO.Stream, y los objetos System.Exception para una funcionalidad de control de errores específica. Solo su imaginación y sentido común limitan estos tipos de casos de uso.

La ampliación de tipos predefinidos puede ser difícil con los tipos `struct`, ya que se pasan en función del valor a los métodos. Eso significa que los cambios en la estructura se realizan en una copia de la misma. Esos cambios dejarán de verse una vez que se salga del método de extensión. A partir de C# 7.2, puede agregar el modificador `ref` al primer argumento de un método de extensión. La adición del modificador `ref` significa que el primer argumento se pasa por referencia. Esto le permite escribir métodos de

extensión que cambian el estado de la estructura que se amplía.

Instrucciones generales

Aunque sigue considerándose preferible agregar la funcionalidad modificando un código del objeto o derivando un nuevo tipo siempre que sea razonable y posible hacerlo, los métodos de extensión se han convertido en una opción fundamental para crear una funcionalidad reutilizable en todo el ecosistema .NET. Para esas ocasiones en las que no cuente con el control del origen original, si un objeto derivado es inadecuado o imposible, o la funcionalidad no se debe exponer más allá de su ámbito aplicable, los métodos de extensión son una opción excelente.

Para obtener más información sobre los tipos derivados, consulte Herencia.

Al usar un método de extensión para ampliar un tipo cuyo código fuente no está bajo su control, se corre el riesgo de que un cambio en la implementación del tipo interrumpa el método de extensión.

Si se implementan métodos de extensión para un tipo determinado, recuerde los puntos siguientes:

- Nunca se llamará a un método de extensión si tiene la misma signatura que un método definido en el tipo.
- Los métodos de extensión se incluyen en el ámbito en el nivel de espacio de nombres. Por ejemplo, si se tienen varias clases estáticas que contienen métodos de extensión en un único espacio de nombres denominado `Extensions`, la directiva `using Extensions;` los incluirá a todos en el ámbito.

Para una biblioteca de clases ya implementada, no deben usarse métodos de extensión para evitar incrementar el número de versión de un ensamblado. Si quiere agregar una funcionalidad significativa a una biblioteca de cuyo código fuente es propietario, siga las instrucciones de .NET estándar para el control de versiones de ensamblado. Para obtener más información, vea Versiones de los ensamblados.

Vea también

- Guía de programación de C#
- Ejemplos de programación en paralelo (incluyen numerosos métodos de extensión de ejemplo)
- Expresiones lambda
- Información general sobre operadores de consulta estándar
- Conversion rules for Instance parameters and their impact (Reglas de conversión para los parámetros de instancia y su impacto)
- Extension methods Interoperability between languages (Interoperabilidad de los métodos de extensión entre lenguajes)
- Extension methods and Curried Delegates (Métodos de extensión y delegados currificados)
- Extension method Binding and Error reporting (Enlazar métodos de extensión y notificación de errores)

Procedimiento para implementar e invocar un método de extensión personalizado (Guía de programación de C#)

29/06/2020 • 4 minutes to read • Edit Online

En este tema se muestra cómo implementar sus propios métodos de extensión para cualquier tipo de .NET. El código de cliente puede usar los métodos de extensión agregando una referencia a la DLL que los contiene y agregando una directiva using que especifique el espacio de nombres en el que se definen los métodos de extensión.

Para definir y llamar al método de extensión

1. Defina una class estática que contenga el método de extensión.

   La clase debe estar visible para el código de cliente. Para obtener más información sobre las reglas de accesibilidad, vea Modificadores de acceso.

2. Implemente el método de extensión como un método estático con al menos la misma visibilidad que la clase contenedora.

3. El primer parámetro del método especifica el tipo en el que opera el método; debe ir precedido del modificador this.

4. En el código de llamada, agregue una directiva `using` para especificar el espacio de nombres que contiene la clase del método de extensión.

5. Llame a los métodos como si fueran métodos de instancia del tipo.

   Tenga en cuenta que el primer parámetro no se especifica mediante el código de llamada, ya que representa el tipo al que se aplica el operador y el compilador ya conoce el tipo del objeto. Solo tiene que proporcionar argumentos para los parámetros comprendidos entre el 2 y `n`.

Ejemplo

En el ejemplo siguiente se implementa un método de extensión denominado `WordCount` en la clase `CustomExtensions.StringExtension`. El método opera en la clase String, que se especifica como el primer parámetro de método. El espacio de nombres `CustomExtensions` se importa en el espacio de nombres de la aplicación y se llama al método dentro del método `Main`.

```
using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this String str)
        {
            return str.Split(new char[] {' ', '.','?'}, StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}
```

Seguridad de .NET

Los métodos de extensión no presentan ninguna vulnerabilidad de seguridad específica. No se pueden usar nunca para suplantar los métodos existentes en un tipo, porque todos los conflictos de nombres se resuelven a favor de la instancia o del método estático definido por el tipo en cuestión. Los métodos de extensión no pueden tener acceso a los datos privados de la clase extendida.

Vea también

- Guía de programación de C#
- Métodos de extensión
- LINQ (Language Integrated Query)
- Clases estáticas y sus miembros
- protected
- internal
- public
- this
- namespace

Procedimiento para crear un método nuevo para una enumeración (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Puede usar métodos de extensión para agregar funcionalidad a un tipo de enumeración concreto.

Ejemplo

En el ejemplo siguiente, la enumeración `Grades` representa las posibles calificaciones con letras que un alumno puede recibir en una clase. Un método de extensión denominado `Passing` se agrega al tipo `Grades` para que cada instancia de ese tipo "sepa" ahora si representa una calificación de aprobado o no.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
        }
    }
  }
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
    Second is not a passing grade.
 */
```

Tenga en cuenta que la clase `Extensions` también contiene una variable estática que se actualiza dinámicamente y

que el valor devuelto del método de extensión refleja el valor actual de esa variable. Esto demuestra que, en segundo plano, los métodos de extensión se invocan directamente en la clase estática en donde se definen.

Vea también

- Guía de programación de C#
- Métodos de extensión

Argumentos opcionales y con nombre (Guía de programación de C#)

18/03/2020 • 13 minutes to read • Edit Online

C# 4 introduce argumentos opcionales y con nombre. Los *argumentos con nombre* permiten especificar un argumento para un parámetro concreto asociando el argumento al nombre del parámetro y no a la posición del parámetro en la lista de parámetros. Los *argumentos opcionales* permiten omitir argumentos para algunos parámetros. Ambas técnicas se pueden usar con métodos, indexadores, constructores y delegados.

Cuando se usan argumentos opcionales y con nombre, los argumentos se evalúan por el orden en que aparecen en la lista de argumentos, no en la lista de parámetros.

Los parámetros opcionales y con nombre, cuando se usan conjuntamente, solo permiten especificar argumentos para algunos parámetros de una lista de parámetros opcionales. Esta funcionalidad facilita enormemente las llamadas a interfaces COM, como las API de automatización de Microsoft Office.

Argumentos con nombre

Los argumentos con nombre le liberan de la necesidad de recordar o buscar el orden de los parámetros de la lista de parámetros de los métodos llamados. El parámetro de cada argumento se puede especificar por nombre de parámetro. Por ejemplo, se puede llamar de la manera habitual a una función que imprime los detalles de un pedido (como por ejemplo, el nombre de vendedor, el nombre de producto y el número del pedido) mediante el envío de argumentos por posición, en el orden definido por la función.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

Si no recuerda el orden de los parámetros pero conoce sus nombres, puede enviar los argumentos en cualquier orden.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
```

```
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Los argumentos con nombre también mejoran la legibilidad del código al identificar lo que cada argumento representa. En el método de ejemplo siguiente, `sellerName` no puede ser nulo ni un espacio en blanco. Como `sellerName` y `productName` son tipos de cadena, en lugar de enviar argumentos por posición, tiene sentido usar argumentos con nombre para eliminar la ambigüedad entre ambos y evitar confusiones para aquellos que lean el código.

Los argumentos con nombre, cuando se usan con argumentos posicionales, son válidos siempre que

- no vayan seguidos de ningún argumento posicional o,

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- *a partir de C# 7.2*, se usen en la posición correcta. En este ejemplo, el parámetro `orderNum` está en la posición correcta pero no se le asigna un nombre de manera explícita.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

Los argumentos posicionales que siguen a los argumentos con nombre desordenados no son válidos.

```
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been
specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Ejemplo

Con este código se implementan los ejemplos de esta sección junto con otros adicionales.

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");    // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                   // C# 7.2 onwards

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or empty.", paramName:
nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product: {productName}");
    }
}
```

Argumentos opcionales

La definición de un método, constructor, indexador o delegado puede especificar que sus parámetros son necesarios o que son opcionales. Todas las llamadas deben proporcionar argumentos para todos los parámetros necesarios, pero pueden omitir los argumentos para los parámetros opcionales.

Cada parámetro opcional tiene un valor predeterminado como parte de su definición. Si no se envía ningún argumento para ese parámetro, se usa el valor predeterminado. Un valor predeterminado debe ser uno de los siguientes tipos de expresiones:

- una expresión constante;
- una expresión con el formato `new ValType()`, donde `ValType` es un tipo de valor, como enum o struct;
- una expresión con el formato default(ValType), donde `ValType` es un tipo de valor.

Los parámetros opcionales se definen al final de la lista de parámetros después de los parámetros necesarios. Si

el autor de la llamada proporciona un argumento para algún parámetro de una sucesión de parámetros opcionales, debe proporcionar argumentos para todos los parámetros opcionales anteriores. No se admiten espacios separados por comas en la lista de argumentos. Por ejemplo, en el código siguiente, el método de instancia `ExampleMethod` se define con un parámetro necesario y dos opcionales.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

La llamada siguiente a `ExampleMethod` genera un error del compilador, porque se proporciona un argumento para el tercer parámetro pero no para el segundo.

```
//anExample.ExampleMethod(3, ,4);
```

Pero si conoce el nombre del tercer parámetro, puede usar un argumento con nombre para realizar la tarea.

```
anExample.ExampleMethod(3, optionalint: 4);
```

En IntelliSense los corchetes se usan para indicar parámetros opcionales, como se muestra en la ilustración siguiente:

> **NOTE**
>
> También puede declarar parámetros opcionales con la clase OptionalAttribute de .NET. Los parámetros `OptionalAttribute` no requieren un valor predeterminado.

Ejemplo

En el ejemplo siguiente, el constructor de `ExampleClass` tiene un solo parámetro, que es opcional. El método de instancia `ExampleMethod` tiene un parámetro necesario, `required`, y dos parámetros opcionales, `optionalstr` y `optionalint`. El código de `Main` muestra las distintas formas en que se pueden invocar el constructor y el método.

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
```

```
            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are optional.
        public void ExampleMethod(int required, string optionalstr = "default string",
            int optionalint = 10)
        {
            Console.WriteLine("{0}: {1}, {2}, and {3}.", _name, required, optionalstr,
                optionalint);
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.
}
```

Interfaces COM

Los argumentos opcionales y con nombre, además de compatibilidad con objetos dinámicos y otros avances, mejoran considerablemente la interoperabilidad con las API de COM, como las API de automatización de Office.

Por ejemplo, el método AutoFormat de la interfaz Range de Microsoft Office Excel tiene siete parámetros, todos ellos opcionales. Estos parámetros se muestran en la ilustración siguiente:

```
excelApp.get_Range("A1", "B4").AutoFormat(
```

```
dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
[object Number = Type.Missing], [object Font = Type.Missing],
[object Alignment = Type.Missing], [object Border = Type.Missing],
[object Pattern = Type.Missing], [object Width = Type.Missing])
```

En C# 3.0 y versiones anteriores, se requiere un argumento para cada parámetro, tal y como se muestra en el ejemplo siguiente.

```
// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

Pero la llamada a `AutoFormat` se puede simplificar considerablemente mediante argumentos opcionales y con nombre, que se introducen en C# 4.0. Los parámetros opcionales y con nombre permiten omitir el argumento de un parámetro opcional si no se quiere cambiar el valor predeterminado del parámetro. En la siguiente llamada, solo se especifica un valor para uno de los siete parámetros.

```
// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

Para obtener más información y ejemplos, vea Procedimiento para usar argumentos opcionales y con nombre en la programación de Office (Guía de programación de C#) y Procedimiento para tener acceso a objetos de interoperabilidad de Office mediante las características de Visual C# (Guía de programación de C#).

Overload Resolution

El uso de argumentos opcionales y con nombre afecta a la resolución de sobrecarga de las maneras siguientes:

- Un método, indexador o constructor es un candidato para la ejecución si cada uno de sus parámetros es opcional o corresponde, por nombre o por posición, a un solo argumento de la instrucción de llamada y el argumento se puede convertir al tipo del parámetro.
- Si se encuentra más de un candidato, se aplican las reglas de resolución de sobrecarga de las conversiones preferidas a los argumentos que se especifican explícitamente. Los argumentos omitidos en parámetros opcionales se ignoran.
- Si dos candidatos se consideran igualmente correctos, la preferencia pasa a un candidato que no tenga parámetros opcionales cuyos argumentos se hayan omitido en la llamada. Se trata de una consecuencia de una preferencia general en la resolución de sobrecarga para los candidatos con menos parámetros.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Procedimiento para usar argumentos opcionales y con nombre en la programación de Office
- Uso de tipo dinámico
- Utilizar constructores
- Utilizar indizadores

Procedimiento para usar argumentos opcionales y con nombre en la programación de Office (Guía de programación de C#)

18/03/2020 • 8 minutes to read • Edit Online

Los argumentos con nombre y los argumentos opcionales, introducidos en C# 4, mejoran la comodidad, la flexibilidad y la legibilidad en la programación de C#. Además, estas características facilitan enormemente el acceso a interfaces COM, como las API de automatización de Microsoft Office.

En el ejemplo siguiente, el método ConvertToTable tiene dieciséis parámetros que representan las características de una tabla, como el número de columnas y filas, el formato, los bordes, las fuentes y los colores. Los dieciséis parámetros son opcionales, ya que la mayoría de las veces no interesa especificar valores concretos para todos ellos. Pero si no hay argumentos opcionales y con nombre, es necesario proporcionar un valor o un valor de marcador de posición para cada parámetro. Con los argumentos opcionales y con nombre, puede especificar valores solamente para los parámetros necesarios para el proyecto.

Debe tener Microsoft Office Word instalado en el equipo para completar estos procedimientos.

> **NOTE**
>
> Es posible que el equipo muestre nombres o ubicaciones diferentes para algunos de los elementos de la interfaz de usuario de Visual Studio en las siguientes instrucciones. La edición de Visual Studio que se tenga y la configuración que se utilice determinan estos elementos. Para obtener más información, vea Personalizar el IDE.

Para crear una aplicación de consola nueva

1. Inicie Visual Studio.
2. En el menú **Archivo** , elija **Nuevo**y haga clic en **Proyecto**.
3. En el panel **Templates Categories** (Categorías de plantillas), expanda **Visual C#** y haga clic en **Windows**.
4. En la parte superior del panel **Plantillas**, asegúrese de que **.NET Framework 4** aparece en el cuadro **Plataforma de destino**.
5. En el panel **Plantillas**, haga clic en **Aplicación de consola**.
6. Escriba un nombre para el proyecto en el campo **Nombre**.
7. Haga clic en **Aceptar**.

   El nuevo proyecto aparece en el **Explorador de soluciones**.

Para agregar una referencia

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el nombre del proyecto y luego haga clic en **Agregar referencia**. Aparecerá el cuadro de diálogo **Agregar referencia**.
2. En la página **.NET**, seleccione **Microsoft.Office.Interop.Word** en la lista **Nombre de componente**.
3. Haga clic en **Aceptar**.

Para agregar las directivas using necesarias

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el archivo *Program.cs* y, después, haga clic en **Ver código**.

2. Agregue las directivas `using` siguientes a la parte superior del archivo de código:

```
using Word = Microsoft.Office.Interop.Word;
```

Para mostrar texto en un documento de Word

1. En la clase `Program` en *Program.cs*, agregue el método siguiente para crear una aplicación de Word y un documento de Word. El método Add tiene cuatro parámetros opcionales. En este ejemplo se usan los valores predeterminados. Por lo tanto, no se necesitan argumentos en la instrucción de llamada.

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

2. Agregue el código siguiente al final del método para definir dónde se muestra texto en el documento y qué texto se muestra:

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

Para ejecutar la aplicación

1. Agregue la instrucción siguiente a Principal:

```
DisplayInWord();
```

2. Presione `CTRL+F5` para ejecutar el proyecto. Aparecerá un documento de Word con el texto especificado.

Para convertir el texto en tabla

1. Use el método `ConvertToTable` para incluir el texto en una tabla. El método tiene 16 parámetros opcionales. IntelliSense coloca los parámetros opcionales entre corchetes, tal como se muestra en la ilustración siguiente.

```
range.ConvertToTable(
```

```
Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

Los argumentos opcionales y con nombre permiten especificar valores solo para los parámetros que quiere cambiar. Agregue el código siguiente al final del método `DisplayInWord` para crear una tabla simple. El argumento especifica que las comas de la cadena de texto de `range` separan las celdas de la tabla.

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

En versiones anteriores de C#, la llamada a `ConvertToTable` requiere un argumento de referencia para cada parámetro, tal como se muestra en el código siguiente:

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code
// is not part of the solution.
var missing = Type.Missing;
object separator = ",";
range.ConvertToTable(ref separator, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing);
```

2. Presione CTRL+F5 para ejecutar el proyecto.

Para experimentar con otros parámetros

1. Para cambiar la tabla de modo que tenga una columna y tres filas, reemplace la última línea de `DisplayInWord` por la instrucción siguiente y escriba CTRL+F5.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. Para especificar un formato predefinido para la tabla, reemplace la última línea de `DisplayInWord` por la instrucción siguiente y escriba CTRL+F5. El formato puede ser cualquiera de las constantes WdTableFormat.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Ejemplo

En el código siguiente se incluye el ejemplo completo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}
```

Vea también

- Argumentos opcionales y con nombre

Constructores (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Cada vez que se crea una clase o struct, se llama a su constructor. Una clase o struct puede tener varios constructores que toman argumentos diferentes. Los constructores permiten al programador establecer valores predeterminados, limitar la creación de instancias y escribir código flexible y fácil de leer. Para obtener más información y ejemplos, vea Usar constructores y Constructores de instancias.

Constructores sin parámetros

Si no proporciona un constructor para la clase, C# creará uno de manera predeterminada que cree instancias del objeto y establezca las variables miembro en los valores predeterminados que se indican en el artículo Valores predeterminados de los tipos de C#. Si no proporciona un constructor para su struct, C# se basa en un *constructor sin parámetros implícito* para inicializar automáticamente cada campo en su valor predeterminado. Para obtener más información y ejemplos, vea Constructores de instancias.

Sintaxis del constructor

Un constructor es un método cuyo nombre es igual que el nombre de su tipo. Su firma del método incluye solo el nombre del método y su lista de parámetros; no incluye un tipo de valor devuelto. En el ejemplo siguiente se muestra el constructor de una clase denominada `Person`.

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }

   // Remaining implementation of Person class.
}
```

Si un constructor puede implementarse como una instrucción única, puede usar una definición del cuerpo de expresión. En el ejemplo siguiente se define una clase `Location` cuyo constructor tiene un único parámetro de cadena denominado *name*. La definición del cuerpo de expresión asigna el argumento al campo `locationName`.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Constructores estáticos

En los ejemplos anteriores se han mostrado constructores de instancia, que crean un objeto nuevo. Una clase o struct también puede tener un constructor estático, que inicializa los miembros estáticos del tipo. Los constructores estáticos no tienen parámetros. Si no proporciona un constructor estático para inicializar los campos estáticos, el compilador de C# inicializa los campos estáticos en su valor predeterminado, tal como se muestra en el artículo Valores predeterminados de los tipos de C#.

En el ejemplo siguiente se usa un constructor estático para inicializar un campo estático.

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Adult()
   {
      minimumAge = 18;
   }

   // Remaining implementation of Adult class.
}
```

También puede definir un constructor estático con una definición de cuerpo de expresión, como se muestra en el ejemplo siguiente.

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

Para obtener más información y ejemplos, vea Constructores estáticos.

En esta sección

Utilizar constructores

Constructores de instancias

Constructores privados

Constructores estáticos

Escritura de un constructor de copia

Vea también

- Guía de programación de C#
- Clases y structs
- Finalizadores

- static
- Why Do Initializers Run In The Opposite Order As Constructors? Part One (¿Por qué los inicializadores se ejecutan en orden contrario a los constructores? Parte uno)

Utilizar constructores (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

Cuando se crea una class o un struct, se llama a su constructor. Los constructores tienen el mismo nombre que la class o el struct y suelen inicializar los miembros de datos del nuevo objeto.

En el ejemplo siguiente, una clase denominada `Taxi` se define mediante un constructor simple. Luego, se crea una instancia de la clase con el operador new. El constructor `Taxi` se invoca con el operador `new` inmediatamente después de asignar memoria para el nuevo objeto.

```
public class Taxi
{
    public bool IsInitialized;
    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

Un constructor que no toma ningún parámetro se denomina *constructor sin parámetros*. Los constructores sin parámetros se invocan cada vez que se crea una instancia de un objeto mediante el operador `new` y no se especifica ningún argumento en `new`. Para obtener más información, vea Instance Constructors (Constructores de instancias [Guía de programación de C#]).

A menos que la clase sea static, las clases sin constructores tienen un constructor público sin parámetros por el compilador de C# con el fin de habilitar la creación de instancias de clase. Para obtener más información, consulte Clases estáticas y sus miembros.

Puede impedir que se cree una instancia de una clase convirtiendo el constructor en privado, de la manera siguiente:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

Para obtener más información, vea Private Constructors (Constructores privados [Guía de programación de C#]).

Los constructores de tipos struct son similares a los constructores de clases, pero `structs` no puede contener un constructor sin parámetros explícito porque el compilador proporciona uno automáticamente. Este constructor inicializa cada campo del `struct` en los valores predeterminados. Pero este constructor sin parámetros solo se invoca si las instancias de `struct` se crean con `new`. Por ejemplo, este código usa el constructor sin parámetros

para Int32, por lo que se tiene la certeza de que el entero se inicializa:

```
int i = new int();
Console.WriteLine(i);
```

Si bien, el siguiente código genera un error del compilador porque no usa `new` y porque intenta usar un objeto que no se ha inicializado:

```
int i;
Console.WriteLine(i);
```

También puede inicializar o asignar los objetos basados en `structs` (incluidos todos los tipos numéricos integrados) y luego usarlos como en el ejemplo siguiente:

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

Por lo que no es necesario llamar al constructor sin parámetros para un tipo de valor.

Tanto las clases como los `structs` pueden definir constructores que toman parámetros. Los constructores que toman parámetros deben llamarse mediante una instrucción `new` o base. Las clases y `structs` también pueden definir varios constructores y no es necesario definir un constructor sin parámetros. Por ejemplo:

```
public class Employee
{
    public int Salary;

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

Esta clase se puede crear mediante cualquiera de las siguientes instrucciones:

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

Un constructor puede usar la palabra clave `base` para llamar al constructor de una clase base. Por ejemplo:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

En este ejemplo, se llama al constructor de la clase base antes de ejecutar el bloque del constructor. La palabra clave `base` puede usarse con o sin parámetros. Los parámetros del constructor se pueden usar como parámetros en `base` o como parte de una expresión. Para obtener más información, vea base.

En una clase derivada, si un constructor de clase base no se llama explícitamente con la palabra clave `base`, se llama implícitamente al constructor sin parámetros, si hay alguno. Esto significa que las siguientes declaraciones del constructor son en efecto iguales:

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

Si una clase base no proporciona un constructor sin parámetros, la clase derivada debe realizar una llamada explícita a un constructor base mediante `base`.

Un constructor puede invocar otro constructor en el mismo objeto mediante la palabra clave this. Igual que `base`, `this` puede usarse con o sin parámetros. Además, los parámetros del constructor están disponibles como parámetros en `this` o como parte de una expresión. Por ejemplo, el segundo constructor del ejemplo anterior se puede reescribir con `this`:

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

El uso de la palabra clave `this` en el ejemplo anterior llama a este constructor:

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

Los constructores se pueden marcar como public, private, protected, internal, protected internal o private protected. Estos modificadores de acceso definen cómo los usuarios de la clase pueden construir la clase. Para obtener más información, vea Modificadores de acceso.

Un constructor puede declararse estático mediante la palabra clave static. Los constructores estáticos se llaman automáticamente, inmediatamente antes de acceder a los campos estáticos y, por lo general, se usan para inicializar miembros de clase estática. Para obtener más información, vea Static Constructors (Constructores estáticos [Guía de programación de C#]).

Especificación del lenguaje C#

Para obtener más información, vea las secciones Constructores de instancia y Constructores estáticos de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Clases y structs
- Constructores
- Finalizadores

Constructores de instancias (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

Los constructores de instancias se usan para crear e inicializar las variables miembro de instancia cuando se usa la expresión new para crear un objeto de una clase. Para inicializar una clase estática, o variables estáticas en una clase no estática, se define un constructor estático. Para obtener más información, vea Static Constructors (Constructores estáticos [Guía de programación de C#]).

En el siguiente ejemplo se muestra un constructor de instancias:

```
class Coords
{
    public int x, y;

    // constructor
    public Coords()
    {
        x = 0;
        y = 0;
    }
}
```

NOTE

Para mayor claridad, esta clase contiene campos públicos. El uso de campos públicos no es una práctica de programación recomendada porque permite que cualquier método de cualquier parte de un programa obtenga acceso sin restricciones ni comprobaciones a los mecanismos internos de un objeto. Los miembros de datos generalmente deberían ser privados y solo se debería tener acceso a ellos a través de las propiedades y métodos de la clase.

Se llama a este constructor de instancias cada vez que se crea un objeto basado en la clase `Coords`. Un constructor como este, que no toma ningún argumento, se denomina *constructor sin parámetros*. Pero a menudo resulta útil proporcionar constructores adicionales. Por ejemplo, se puede agregar un constructor a la clase `Coords` que permita especificar los valores iniciales de los miembros de datos:

```
// A constructor with two arguments.
public Coords(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

Esto permite crear objetos `Coords` con valores iniciales predeterminados o específicos, como este:

```
var p1 = new Coords();
var p2 = new Coords(5, 3);
```

Si una clase no tiene un constructor, se genera automáticamente un constructor sin parámetros y los valores predeterminados se usan para inicializar los campos del objeto. Por ejemplo, un int se inicializa en 0. Para obtener información sobre los valores predeterminados de los tipos, vea Valores predeterminados de los tipos

de C#. Por tanto, dado que el constructor sin parámetros de la clase `Coords` inicializa todos los miembros de datos en cero, se puede quitar por completo sin cambiar el funcionamiento de la clase. Más adelante en este tema se proporciona un ejemplo completo del uso de varios constructores en el Ejemplo 1 y en el Ejemplo 2 se proporciona un ejemplo de un constructor generado automáticamente.

Los constructores de instancias también se pueden usar para llamar a los constructores de instancias de las clases base. El constructor de clase puede invocar el constructor de la clase base a través del inicializador, como sigue:

```
class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
}
```

En este ejemplo, la clase `Circle` pasa valores que representan el radio y el alto al constructor proporcionado por `Shape` desde el que se deriva `Circle`. Un ejemplo completo del uso de `Shape` y `Circle` aparece en este tema en el Ejemplo 3.

Ejemplo 1

En el ejemplo siguiente se muestra una clase con dos constructores de clase, uno sin argumentos y uno con dos argumentos.

```
class Coords
{
    public int x, y;

    // Default constructor.
    public Coords()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments.
    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Override the ToString method.
    public override string ToString()
    {
        return $"({x},{y})";
    }
}

class MainClass
{
    static void Main()
    {
        var p1 = new Coords();
        var p2 = new Coords(5, 3);

        // Display the results using the overriden ToString method.
        Console.WriteLine($"Coords #1 at {p1}");
        Console.WriteLine($"Coords #2 at {p2}");
        Console.ReadKey();
    }
}
/* Output:
 Coords #1 at (0,0)
 Coords #2 at (5,3)
*/
```

Ejemplo 2

En este ejemplo, la clase `Person` no tiene ningún constructor, en cuyo caso, se proporciona automáticamente un constructor sin parámetros y los campos se inicializan en sus valores predeterminados.

```
public class Person
{
    public int age;
    public string name;
}

class TestPerson
{
    static void Main()
    {
        var person = new Person();

        Console.WriteLine("Name: {person.name}, Age: {person.age}");
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:  Name: , Age: 0
```

Observe que el valor predeterminado de `age` es `0` y el valor predeterminado de `name` es `null`.

Ejemplo 3

En el ejemplo siguiente se muestra cómo usar el inicializador de la clase base. La clase `Circle` se deriva de la clase general `Shape` y la clase `Cylinder` se deriva de la clase `Circle`. En cada clase derivada, el constructor usa su inicializador de clase base.

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
    public override double Area()
    {
        return pi * x * x;
    }
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}

class TestShapes
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);

        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area of the circle = 19.63
    Area of the cylinder = 86.39
*/
```

Para obtener más ejemplos sobre cómo invocar los constructores de clase base, vea virtual, override y base.

Vea también

- Guía de programación de C#
- Clases y structs
- Constructores
- Finalizadores
- static

Constructores privados (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

Un constructor privado es un constructor de instancia especial. Se usa generalmente en clases que contienen solo miembros estáticos. Si una clase tiene uno o más constructores privados y ningún constructor público, el resto de clases (excepto las anidadas) no podrán crear instancias de esta clase. Por ejemplo:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

La declaración de un constructor vacío evita la generación automática de un constructor sin parámetros. Observe que si no usa un modificador de acceso en el constructor, este será privado de manera predeterminada. En cambio, normalmente se usa el modificador private de manera explícita para aclarar que no es posible crear una instancia de la clase.

Los constructores privados se usan para evitar la creación de instancias de una clase cuando no hay campos o métodos de instancia, por ejemplo, la clase Math, o cuando se llama a un método para obtener una instancia de una clase. Si todos los métodos de la clase son estáticos, considere convertir la clase completa en estática. Para obtener más información, vea Clases estáticas y sus miembros.

Ejemplo

El siguiente es un ejemplo de clase que usa un constructor privado.

```
public class Counter
{
    private Counter() { }
    public static int currentCount;
    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Observe que si quita el comentario de la siguiente instrucción del ejemplo, se producirá un error porque el constructor es inaccesible debido a su nivel de protección:

```
// Counter aCounter = new Counter();   // Error
```

Especificación del lenguaje C#

Para obtener más información, vea la sección Constructores privados de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Clases y structs
- Constructores
- Finalizadores
- private
- public

Constructores estáticos (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

Un constructor estático se usa para inicializar cualquier dato estático o realizar una acción determinada que solo debe realizarse una vez. Es llamado automáticamente antes de crear la primera instancia o de hacer referencia a cualquier miembro estático.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Comentarios

Los constructores estáticos tienen las propiedades siguientes:

- Un constructor estático no permite modificadores de acceso ni tiene parámetros.
- Una clase o struct solo puede tener un constructor estático.
- Los constructores estáticos no se pueden heredar ni sobrecargar.
- No se puede llamar a un constructor estático directamente y solo está pensado para que Common Language Runtime (CLR) lo llame. Se invoca automáticamente.
- El usuario no puede controlar cuándo se ejecuta el constructor estático en el programa.
- Se le llama automáticamente para inicializar la clase antes de crear la primera instancia o de hacer referencia a cualquier miembro estático. Un constructor estático se ejecutará antes que un constructor de instancia. Se llama al constructor estático de un tipo cuando se invoca un método estático asignado a un evento o un delegado y no cuando este se asigna. Si los inicializadores de variable del campo estático están presentes en la clase o el constructor estático, se ejecutarán en el orden textual en el que aparecen en la declaración de clase inmediatamente anterior a la ejecución del constructor estático.
- Si no proporciona un constructor estático para inicializar los campos estáticos, todos los campos estáticos se inicializan en su valor predeterminado como se muestra en Valores predeterminados de los tipos de C#.
- Si un constructor estático inicia una excepción, el motor en tiempo de ejecución no lo invocará una segunda vez y el tipo seguirá sin inicializar durante el período de duración del dominio de aplicación donde se ejecuta el programa. Normalmente, se inicia una excepción TypeInitializationException cuando un constructor estático no puede crear una instancia de un tipo o para una excepción no controlada que se produce dentro de un constructor estático. En el caso de los constructores estáticos implícitos no definidos de forma explícita en el código fuente, la solución de problemas puede requerir la inspección

del código de lenguaje intermedio (IL).

- La presencia de un constructor estático evita la adición del atributo de tipo BeforeFieldInit. Esto limita la optimización en tiempo de ejecución.
- Un campo declarado como `static readonly` solo se puede asignar como parte de su declaración o en un constructor estático. Si no se necesita un constructor estático explícito, inicialice campos estáticos en la declaración, en lugar de a través de un constructor estático para una mejor optimización en tiempo de ejecución.

> **NOTE**
> Aunque no es directamente accesible, la presencia de un constructor estático explícito debe documentarse para ayudar con la solución de problemas de excepciones de inicialización.

Uso

- Los constructores estáticos se usan normalmente cuando la clase hace uso de un archivo de registro y el constructor escribe entradas en dicho archivo.
- Los constructores estáticos también son útiles al crear clases contenedoras para código no administrado, cuando el constructor puede llamar al método `LoadLibrary`.
- Los constructores estáticos también son un lugar adecuado para aplicar comprobaciones en tiempo de ejecución en el parámetro de tipo que no se puede comprobar en tiempo de compilación a través de restricciones (restricciones de parámetro de tipo).

Ejemplo

En este ejemplo, la clase `Bus` tiene un constructor estático. Cuando se crea la primera instancia de `Bus` (`bus1`), se invoca el constructor estático para inicializar la clase. En el resultado del ejemplo, se comprueba que el constructor estático se ejecuta solo una vez, incluso si se crean dos instancias de `Bus`, y que se ejecuta antes de que se ejecute el constructor de instancia.

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to {0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }
```

```
    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to simulate
        // actual bus times. Do not do this in your actual bus schedule program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",
                          this.RouteNumber,
                          elapsedTime.Milliseconds,
                          globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

Especificación del lenguaje C#

Para obtener más información, consulte la sección sobre constructores estáticos de la Especificación del lenguaje C#.

Vea también

- Guía de programación de C#
- Clases y structs
- Constructores
- Clases estáticas y sus miembros
- Finalizadores
- Instrucciones de diseño de constructores
- Advertencia de seguridad - CA2121: Los constructores estáticos deben ser privados

Procedimiento para escribir un constructor de copia (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

C# no proporciona un constructor de copias para los objetos, pero puede escribir uno personalmente.

Ejemplo

En el ejemplo siguiente, `Person` class define un constructor de copias que toma, como argumento, una instancia de `Person`. Los valores de las propiedades de los argumentos se asignan a las propiedades de la nueva instancia de `Person`. El código contiene un constructor de copias alternativo que envía las propiedades `Name` y `Age` de la instancia que quiere copiar al constructor de instancia de la clase.

```
class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

Vea también

- ICloneable

- Guía de programación de C#
- Clases y structs
- Constructores
- Finalizadores

Finalizadores (Guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

Los finalizadores (también denominados **destructores**) se usan para realizar cualquier limpieza final necesaria cuando el recolector de elementos no utilizados recopila una instancia de clase.

Comentarios

- Los finalizadores no se pueden definir en structs. Solo se usan con clases.
- Una clase solo puede tener un finalizador.
- Los finalizadores no se pueden heredar ni sobrecargar.
- No se puede llamar a los finalizadores. Se invocan automáticamente.
- Un finalizador no permite modificadores ni tiene parámetros.

Por ejemplo, el siguiente código muestra una declaración de un finalizador para la clase `Car`.

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

Un finalizador también puede implementarse como una definición de cuerpo de expresión, como se muestra en el ejemplo siguiente.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

El finalizador llama implícitamente a Finalize en la clase base del objeto. Por lo tanto, una llamada a un finalizador se convierte implícitamente al siguiente código:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

Este diseño significa que se realizan llamadas al método `Finalize` de manera recursiva para todas las instancias de la cadena de herencia, desde la más a la menos derivada.

> **NOTE**
>
> Los finalizadores vacíos no deben usarse. Cuando una clase contiene un finalizador, se crea una entrada en la cola `Finalize`. Cuando se llama al finalizador, se invoca al recolector de elementos no utilizados para procesar la cola. Un finalizador vacío simplemente produce una pérdida de rendimiento innecesaria.

El programador no puede controlar cuándo se llama al finalizador; es el recolector de elementos no utilizados el que decide cuándo hacerlo. El recolector de elementos no utilizados comprueba si hay objetos que ya no están siendo usados por ninguna aplicación. Si considera un objeto elegible para su finalización, llama al finalizador (si existe) y reclama la memoria usada para almacenar el objeto.

En las aplicaciones de .NET Framework (pero no en las de .NET Core), cuando se cierra el programa también se llama a los finalizadores.

Es posible forzar la recolección de elementos no utilizados si se llama a Collect, pero debe evitarse esta llamada porque puede dar lugar a problemas de rendimiento.

Uso de finalizadores para liberar recursos

En general, C# no requiere tanta administración de memoria por parte del desarrollador como los lenguajes que no están diseñados para un runtime con recolección de elementos no utilizados. Esto es debido a que el recolector de elementos no utilizados de .NET administra implícitamente la asignación y liberación de memoria para los objetos. En cambio, cuando la aplicación encapsule recursos no administrados, como ventanas, archivos y conexiones de red, debería usar finalizadores para liberar dichos recursos. Cuando el objeto cumple los requisitos para su finalización, el recolector de elementos no utilizados ejecuta el método `Finalize` del objeto.

Liberación explícita de recursos

Si la aplicación usa un recurso externo costoso, también es recomendable liberar explícitamente el recurso antes de que el recolector de elementos no utilizados libere el objeto. Para liberar el recurso, implemente un método `Dispose` desde la interfaz IDisposable que realiza la limpieza necesaria del objeto. Esto puede mejorar considerablemente el rendimiento de la aplicación. Aun con este control explícito sobre los recursos, el finalizador se convierte en una salvaguarda para limpiar recursos si se produce un error en la llamada al método `Dispose`.

Para obtener más información sobre la limpieza de recursos, vea los siguientes artículos:

- Limpieza de recursos no administrados
- Implementación de un método Dispose
- using (instrucción)

Ejemplo

En el siguiente ejemplo se crean tres clases que forman una cadena de herencia. La clase `First` es la clase base, `Second` se deriva de `First` y `Third` se deriva de `Second`. Los tres tienen finalizadores. En `Main`, se crea una instancia de la clase más derivada. Cuando ejecute el programa, observe que se llama a los finalizadores de las tres clases automáticamente y en orden, desde la más derivada hasta la menos derivada.

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's destructor is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's destructor is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's destructor is called.");
    }
}

class TestDestructors
{
    static void Main()
    {
        Third t = new Third();
    }
}
/* Output (to VS Output Window):
    Third's destructor is called.
    Second's destructor is called.
    First's destructor is called.
*/
```

Especificación del lenguaje C#

Para más información, vea la sección sobre destructores de la Especificación del lenguaje C#.

Vea también

- IDisposable
- Guía de programación de C#
- Constructores
- Recolección de elementos no utilizados

Inicializadores de objeto y de colección (Guía de programación de C#)

18/03/2020 • 13 minutes to read • Edit Online

C# permite crear instancias de un objeto o colección y realizar asignaciones de miembros en una sola instrucción.

Inicializadores de objeto

Los inicializadores de objeto permiten asignar valores a cualquier campo o propiedad accesible de un objeto en el momento de su creación sin tener que invocar un constructor seguido de líneas de instrucciones de asignación. La sintaxis de inicializador de objetos permite especificar argumentos para un constructor u omitir los argumentos (y la sintaxis de paréntesis). En el ejemplo siguiente se muestra cómo usar un inicializador de objeto con un tipo con nombre, `Cat`, y cómo invocar el constructor sin parámetros. Tenga en cuenta el uso de propiedades implementadas automáticamente en la clase `Cat`. Para obtener más información, vea Auto-Implemented Properties (Propiedades implementadas automáticamente).

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

La sintaxis de los inicializadores de objeto le permite crear una instancia, y después asigna el objeto recién creado, con las propiedades asignadas, a la variable de la asignación.

A partir de C# 6, los inicializadores de objeto pueden establecer indizadores, además de asignar campos y propiedades. Tenga en cuenta esta clase básica `Matrix`:

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

Se podría inicializar la matriz de identidad con el código siguiente:

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Puede usarse cualquier indizador accesible que contenga un establecedor accesible como una de las expresiones de un inicializador de objeto, independientemente del número o los tipos de argumentos. Los argumentos del índice forman el lado izquierdo de la asignación, mientras que el valor es el lado derecho de la expresión. Por ejemplo, todos estos son válidos si `IndexersExample` tiene los indizadores adecuados:

```
var thing = new IndexersExample {
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

Para que el código anterior se compile, el tipo `IndexersExample` debe tener los siguientes miembros:

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] {  set { ... }; }
```

Inicializadores de objeto con tipos anónimos

Aunque los inicializadores de objeto se pueden usar en cualquier contexto, resultan especialmente útiles en las expresiones de consulta LINQ. Las expresiones de consulta usan con frecuencia tipos anónimos, que solo se pueden inicializar con un inicializador de objeto, como se muestra en la siguiente declaración.

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Los tipos anónimos permiten a la cláusula `select` de una expresión de consulta LINQ transformar objetos de la secuencia original en objetos cuyo valor y forma pueden ser distintos de los originales. Esto resulta útil si desea almacenar solo una parte de la información de cada objeto en una secuencia. En el ejemplo siguiente, suponga que un objeto del producto (`p`) contiene numerosos campos y métodos y que solo le interesa crear una secuencia de objetos que contenga el nombre del producto y el precio por unidad.

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

Al ejecutarse esta consulta, la variable `productInfos` incluirá una secuencia de objetos a la que se puede tener acceso en una instrucción `foreach`, como se muestra en este ejemplo:

```
foreach(var p in productInfos){...}
```

Cada objeto del nuevo tipo anónimo tiene dos propiedades públicas que reciben los mismos nombres que las propiedades o los campos del objeto original. También puede cambiar el nombre de un campo al crear un tipo anónimo; en el ejemplo siguiente se cambia el nombre del campo `UnitPrice` a `Price`.

```
select new {p.ProductName, Price = p.UnitPrice};
```

Inicializadores de colección

Los inicializadores de colección le permiten especificar uno o varios inicializadores de elemento al inicializar un tipo de colección que implementa IEnumerable y tiene `Add` con la firma apropiada como un método de instancia o un método de extensión. Los inicializadores de elemento pueden ser un valor simple, una expresión o un inicializador de objeto. Si se usa un inicializador de colección, no es necesario especificar varias llamadas; el compilador las agrega automáticamente.

En el ejemplo siguiente se muestran dos inicializadores de colección simples:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

El inicializador de colección siguiente usa inicializadores de objeto para inicializar los objetos de la clase `Cat` definida en un ejemplo anterior. Observe que los inicializadores de objeto individuales se escriben entre llaves y se separan por comas.

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

Puede especificar null como elemento de un inicializador de colección si el método `Add` de la colección lo permite.

```
List<Cat> moreCats = new List<Cat>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

Puede especificar elementos indexados si la colección admite indexación de lectura y escritura.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

El ejemplo anterior genera código que llama a Item[TKey] para establecer los valores. A partir de C# 6, puede inicializar diccionarios y otros contenedores asociativos con la sintaxis siguiente. Tenga en cuenta que en lugar de sintaxis de indizador, con paréntesis y una asignación, usa un objeto con varios valores:

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

Este ejemplo de inicializador llama a Add(TKey, TValue) para agregar los tres elementos al diccionario. Estas dos maneras distintas de inicializar colecciones asociativas tienen un comportamiento ligeramente diferente debido a las llamadas a métodos que genera el compilador. Ambas variantes funcionan con la clase `Dictionary`. Es posible que otros tipos solo admitan una o la otra, en función de su API pública.

Ejemplos

En el ejemplo siguiente se combinan los conceptos de inicializadores de objeto y colección.

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

El ejemplo siguiente muestra un objeto que implementa IEnumerable y que contiene un método `Add` con varios parámetros. Usa un inicializador de colección con varios elementos por cada elemento de la lista correspondiente a la firma del método `Add`.

```
    public class FullExample
    {
        class FormattedAddresses : IEnumerable<string>
        {
            private List<string> internalList = new List<string>();
            public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

            public void Add(string firstname, string lastname,
                string street, string city,
                string state, string zipcode) => internalList.Add(
                $@"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
                );
        }

        public static void Main()
        {
            FormattedAddresses addresses = new FormattedAddresses()
            {
                {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
                {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
            };

            Console.WriteLine("Address Entries:");

            foreach (string addressEntry in addresses)
            {
                Console.WriteLine("\r\n" + addressEntry);
            }
        }

        /*
         * Prints:

            Address Entries:

            John Doe
            123 Street
            Topeka, KS 00000

            Jane Smith
            456 Street
            Topeka, KS 00000
         */
    }
```

Los métodos `Add` pueden usar la palabra clave `params` para tomar un número variable de argumentos, como se muestra en el ejemplo siguiente. En este ejemplo además se muestra la implementación personalizada de un indizador para inicializar una colección mediante índices.

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();
```

```
        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key, (IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue> storedValues))
                internalDictionary.Add(key, storedValues = new List<TValue>());

            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }
            };

        Console.WriteLine("Using first multi-valued dictionary created with a collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");
```

```
            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
     * Prints:

        Using first multi-valued dictionary created with a collection initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
     */
}
```

Vea también

- Guía de programación de C#
- LINQ en C#
- Tipos anónimos

Procedimiento para inicializar objetos usando un inicializador de objeto (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

Puede usar inicializadores de objeto para inicializar objetos de tipo de una forma declarativa sin tener que invocar explícitamente un constructor para el tipo.

En los siguientes ejemplos se muestra cómo usar los inicializadores de objeto con objetos con nombre. El compilador procesa los inicializadores de objeto primero obteniendo acceso al constructor de instancia predeterminado y después procesando las inicializaciones de miembro. Por lo tanto, si el constructor sin parámetros se declara como `private` en la clase, se producirá un error en los inicializadores de objeto que requieren acceso público.

Debe usar un inicializador de objeto si va a definir un tipo anónimo. Para obtener más información, vea Procedimiento para devolver subconjuntos de propiedades de elementos en una consulta.

Ejemplo

En el siguiente ejemplo se muestra cómo inicializar un nuevo tipo `StudentName` usando inicializadores de objeto. Este ejemplo establece propiedades en el tipo `StudentName`:

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The default constructor is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the default constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };
```

```
        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // The default constructor has no parameters. The default constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "  " + ID;
    }
}
```

Los inicializadores de objeto pueden usarse para establecer indizadores en un objeto. En el ejemplo siguiente se define una clase `BaseballTeam` que usa un indizador para obtener y establecer jugadores en posiciones diferentes. El inicializador puede asignar jugadores en función de la abreviatura de la posición o del número usado para las puntuaciones de béisbol de cada posición:

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

Vea también

- Guía de programación de C#
- Inicializadores de objeto y colección

Procedimientos: inicialización de un diccionario con un inicializador de colección (guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Una clase Dictionary<TKey,TValue> contiene una colección de pares clave-valor. Su método Add toma dos parámetros: uno para la clave y otro para el valor. Una manera de inicializar Dictionary<TKey,TValue>, o cualquier colección cuyo método `Add` tome varios parámetros, es incluir entre llaves cada conjunto de parámetros, como se muestra en el ejemplo siguiente. Otra opción es usar un inicializador de índice, lo que también se muestra en el ejemplo siguiente.

Ejemplo

En el ejemplo de código siguiente, Dictionary<TKey,TValue> se inicializa con instancias de tipo `StudentName`. La primera inicialización usa el método `Add` con dos argumentos. El compilador genera una llamada a `Add` por cada uno de los pares de claves `int` y valores `StudentName`. La segunda usa un método de indizador de lectura y escritura público de la clase `Dictionary`:

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students2[index].FirstName} {students2[index].LastName}");
        }
    }
}
```

Observe los dos pares de llaves de cada elemento de la colección en la primera declaración. Las llaves internas contienen el inicializador de objeto para `StudentName`, mientras que las externas contienen el inicializador para el par clave-valor que se va a agregar a la clase `students` Dictionary<TKey,TValue>. Por último, el inicializador completo de la colección para el diccionario se encierra entre llaves. En la segunda inicialización, el lado izquierdo de la asignación es la clave y el lado derecho es el valor, con un inicializador de objeto para `StudentName`.

Vea también

- Guía de programación de C#
- Inicializadores de objeto y colección

Tipos anidados (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Un tipo definido en una clase, estructura o interfaz se denomina tipo anidado. Por ejemplo

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Con independencia de si el tipo externo es una clase, una interfaz o una estructura, los tipos anidados se establecen de manera predeterminada en private; solo son accesibles desde su tipo contenedor. En el ejemplo anterior, la clase `Nested` es inaccesible a los tipos externos.

También puede especificar un modificador de acceso para definir la accesibilidad de un tipo anidado, de la manera siguiente:

- Los tipos anidados de una **clase** pueden ser public, protected, internal, protected internal, private o private protected.

  En cambio, al definir una clase anidada `protected`, `protected internal` o `private protected` dentro de una clase sellada, se genera una advertencia del compilador CS0628, "Nuevo miembro protegido declarado en la clase sealed".

- Los tipos anidados de un **struct** pueden ser public, internal o private.

En el ejemplo siguiente se convierte la clase `Nested` en public:

```
public class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

El tipo anidado o interno puede tener acceso al tipo contenedor o externo. Para tener acceso al tipo contenedor, páselo como un argumento al constructor del tipo anidado. Por ejemplo:

```
public class Container
{
    public class Nested
    {
        private Container parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

Un tipo anidado tiene acceso a todos los miembros que estén accesibles para el tipo contenedor. Puede tener acceso a los miembros privados y protegidos del tipo contenedor, incluidos los miembros protegidos heredados.

En la declaración anterior, el nombre completo de la clase `Nested` es `Container.Nested`. Este es el nombre que se utiliza para crear una instancia nueva de la clase anidada, de la siguiente manera:

```
Container.Nested nest = new Container.Nested();
```

Vea también

- Guía de programación de C#
- Clases y structs
- Modificadores de acceso
- Constructores

Clases y métodos parciales (Guía de programación de C#)

18/03/2020 • 12 minutes to read • Edit Online

Es posible dividir la definición de una clase, un struct, una interfaz o un método en dos o más archivos de código fuente. Cada archivo de código fuente contiene una sección de la definición de tipo o método, y todos los elementos se combinan cuando se compila la aplicación.

Clases parciales

Es recomendable dividir una definición de clase en varias situaciones:

- Cuando se trabaja con proyectos grandes, el hecho de repartir una clase entre archivos independientes permite que varios programadores trabajen en ella al mismo tiempo.
- Cuando se trabaja con código fuente generado automáticamente, se puede agregar código a la clase sin tener que volver a crear el archivo de código fuente. Visual Studio usa este enfoque al crear formularios Windows Forms, código de contenedor de servicio Web, etc. Puede crear código que use estas clases sin necesidad de modificar el archivo creado por Visual Studio.
- Para dividir una definición de clase, use el modificador de palabra clave partial, como se muestra aquí:

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

La palabra clave `partial` indica que se pueden definir en el espacio de nombres otros elementos de la clase, la estructura o la interfaz. Todos los elementos deben usar la palabra clave `partial`. Todos los elementos deben estar disponibles en tiempo de compilación para formar el tipo final. Todos los elementos deben tener la misma accesibilidad, como `public`, `private`, etc.

Si algún elemento se declara abstracto, todo el tipo se considera abstracto. Si algún elemento se declara sellado, todo el tipo se considera sellado. Si algún elemento declara un tipo base, todo el tipo hereda esa clase.

Todos los elementos que especifiquen una clase base deben coincidir, pero los elementos que omitan una clase base heredan igualmente el tipo base. Los elementos pueden especificar diferentes interfaces base, y el tipo final implementa todas las interfaces enumeradas por todas las declaraciones parciales. Todas las clases, structs o miembros de interfaz declarados en una definición parcial están disponibles para todos los demás elementos. El tipo final es la combinación de todos los elementos en tiempo de compilación.

> **NOTE**
> El modificador `partial` no está disponible en declaraciones de delegado o enumeración.

En el ejemplo siguiente se muestra que los tipos anidados pueden ser parciales, incluso si el tipo en el que están anidados no es parcial.

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }
    partial class Nested
    {
        void Test2() { }
    }
}
```

En tiempo de compilación, se combinan los atributos de definiciones de tipo parcial. Por ejemplo, consideremos las siguientes declaraciones:

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

Son equivalentes a las declaraciones siguientes:

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

A continuación se indican los elementos que se combinan de todas las definiciones de tipo parcial:

- comentarios XML
- interfaces
- atributos de parámetro de tipo genérico
- class (atributos)
- miembros

Por ejemplo, consideremos las siguientes declaraciones:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

Son equivalentes a las declaraciones siguientes:

```
class Earth : Planet, IRotate, IRevolve { }
```

Restricciones

Debe seguir varias reglas al trabajar con definiciones de clase parcial:

- Todas las definiciones de tipo parcial que van a formar parte del mismo tipo deben modificarse con `partial`. Por ejemplo, las declaraciones de clase siguientes generan un error:

  ```
  public partial class A { }
  //public class A { }  // Error, must also be marked partial
  ```

- El modificador `partial` solo puede aparecer inmediatamente antes de las palabras clave `class`, `struct` o `interface`.

- Se permiten tipos parciales anidados en definiciones de tipo parcial, como se muestra en el ejemplo siguiente:

  ```
  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }

  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }
  ```

- Todas las definiciones de tipo parcial que van a formar parte del mismo tipo deben definirse en el mismo ensamblado y en el mismo módulo (archivo .exe o .dll). Las definiciones parciales no pueden abarcar varios módulos.

- El nombre de clase y los parámetros de tipo genérico deben coincidir en todas las definiciones de tipo parcial. Los tipos genéricos pueden ser parciales. Cada declaración parcial debe usar los mismos nombres de parámetro en el mismo orden.

- Las siguientes palabras clave son opcionales en una definición de tipo parcial, pero si están presentes la definición, no pueden entrar en conflicto con las palabras clave especificadas en otra definición parcial para el mismo tipo:

  - public
  - private
  - protected
  - internal
  - abstract
  - sealed
  - clase base
  - modificador new (elementos anidados)
  - restricciones genéricas

Para obtener más información, vea Restricciones de tipos de parámetros.

Ejemplo 1

Descripción

En el ejemplo siguiente, los campos y el constructor de la clase, `Coords`, se declaran en una definición de clase parcial y el miembro `PrintCoords` se declara en otra definición de clase parcial.

Código

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

Ejemplo 2

Descripción

En el ejemplo siguiente se muestra que también se pueden desarrollar structs e interfaces parciales.

Código

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Métodos Partial

Una clase o struct parcial puede contener un método parcial. Un elemento de la clase contiene la firma del método. Se puede definir una implementación opcional en el mismo elemento o en otro. Si no se proporciona la implementación, el método y todas las llamadas al método se quitan en tiempo de compilación.

Los métodos parciales permiten que el implementador de un elemento de una clase defina un método, similar a un evento. El implementador del otro elemento de la clase puede decidir si quiere implementar el método o no. Si el método no se implementa, el compilador quita la firma del método y todas las llamadas al método. Las llamadas al método, incluidos los resultados que se producirían por la evaluación de los argumentos de las llamadas, no tienen efecto en tiempo de ejecución. Por lo tanto, el código de la clase parcial puede usar libremente un método parcial, incluso si no se proporciona la implementación. No se producirá ningún error en tiempo de compilación o en tiempo de ejecución si se llama al método pero no se implementa.

Los métodos parciales son especialmente útiles para personalizar el código generado. Permiten reservar un nombre y firma de método de modo que el código generado pueda llamar al método, pero el desarrollador decide si se implementa el método. De manera muy similar a como hacen las clases parciales, los métodos parciales permiten que el código creado por un generador de código y el código creado por un desarrollador humano funcionen juntos sin costos en tiempo de ejecución.

Una declaración de método parcial consta de dos elementos: la definición y la implementación. Pueden encontrarse en elementos independientes de una clase parcial o en el mismo elemento. Si no hay ninguna declaración de implementación, el compilador optimiza tanto la declaración de definición como todas las llamadas al método.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

- Las declaraciones de método parcial deben comenzar con la palabra clave contextual partial y el método debe devolver void.
- Los métodos parciales pueden tener parámetros in o ref, pero no parámetros out.

- Los métodos parciales son implícitamente private y, por tanto, no pueden ser virtual.
- Los métodos parciales no pueden ser extern, ya que la presencia del cuerpo determina si son de definición o de implementación.
- Los métodos parciales pueden tener modificadores static y unsafe.
- Los métodos parciales pueden ser genéricos. Las restricciones se colocan en la declaración de método parcial de definición y opcionalmente pueden repetirse en el de implementación. Los nombres del parámetro y del parámetro de tipo no tienen que ser iguales en la declaración de implementación y en la declaración de definición.
- Puede crear un delegado para un método parcial que se ha definido e implementado, pero no para un método parcial que solo se ha definido.

Especificación del lenguaje C#

Para obtener más información, vea la sección Tipos parciales de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Clases
- Tipos de estructura
- Interfaces
- partial (Tipos)

Tipos anónimos (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

Los tipos anónimos son una manera cómoda de encapsular un conjunto de propiedades de solo lectura en un único objeto sin tener que definir primero un tipo explícitamente. El compilador genera el nombre del tipo y no está disponible en el nivel de código fuente. El compilador deduce el tipo de cada propiedad.

Para crear tipos anónimos, use el operador new con un inicializador de objeto. Para obtener más información sobre los inicializadores de objeto, vea Inicializadores de objeto y colección (Guía de programación de C#).

En el ejemplo siguiente se muestra un tipo anónimo que se inicializa con dos propiedades llamadas `Amount` y `Message`.

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Los tipos anónimos suelen usarse en la cláusula select de una expresión de consulta para devolver un subconjunto de las propiedades de cada objeto en la secuencia de origen. Para más información sobre las consultas, vea LINQ en C#.

Los tipos anónimos contienen una o varias propiedades públicas de solo lectura. No es válido ningún otro tipo de miembros de clase, como métodos o eventos. La expresión que se usa para inicializar una propiedad no puede ser `null`, una función anónima o un tipo de puntero.

El escenario más habitual es inicializar un tipo anónimo con propiedades de otro tipo. En el siguiente ejemplo, se da por hecho que existe una clase con el nombre `Product`. La clase `Product` incluye las propiedades `Color` y `Price`, junto con otras propiedades que no son de su interés. La variable `products` es una colección de objetos `Product`. La declaración de tipo anónimo comienza con la palabra clave `new`. La declaración inicializa un nuevo tipo que solo usa dos propiedades de `Product`. Esto hace que la consulta devuelva una cantidad de datos menor.

Si no especifica los nombres de miembro en el tipo anónimo, el compilador da a los miembros de tipo anónimo el mismo nombre que la propiedad que se usa para inicializarlos. Debe proporcionar un nombre para una propiedad que se está inicializando con una expresión, como se muestra en el ejemplo anterior. En el siguiente ejemplo, los nombres de las propiedades del tipo anónimo son `Color` y `Price`.

```
var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

Normalmente, cuando se usa un tipo anónimo para inicializar una variable, la variable se declara como variable local con tipo implícito mediante var. El nombre del tipo no se puede especificar en la

declaración de la variable porque solo el compilador tiene acceso al nombre subyacente del tipo anónimo. Para obtener más información sobre `var`, vea Variables locales con asignación implícita de tipos.

Puede crear una matriz de elementos con tipo anónimo combinando una variable local con tipo implícito y una matriz con tipo implícito, como se muestra en el ejemplo siguiente.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

Comentarios

Los tipos anónimos son tipos class que derivan directamente de object y que no se pueden convertir a ningún tipo excepto object. El compilador proporciona un nombre para cada tipo anónimo, aunque la aplicación no pueda acceder a él. Desde el punto de vista de Common Language Runtime, un tipo anónimo no es diferente de otros tipos de referencia.

Si dos o más inicializadores de objeto anónimo en un ensamblado especifican una secuencia de propiedades que están en el mismo orden y que tienen los mismos nombres y tipos, el compilador trata el objeto como instancias del mismo tipo. Comparten la misma información de tipo generada por el compilador.

No se puede declarar que un campo, una propiedad, un evento o el tipo de valor devuelto de un método tengan un tipo anónimo. De forma similar, no se puede declarar que un parámetro formal de un método, propiedad, constructor o indizador tenga un tipo anónimo. Para pasar un tipo anónimo, o una colección que contiene tipos anónimos, como un argumento a un método, puede declarar el parámetro como objeto de tipo. Sin embargo, al hacerlo se invalida el propósito del tipado fuerte. Si tiene que almacenar resultados de consulta o pasarlos fuera del límite del método, considere la posibilidad de usar un struct o una clase con nombre normal en lugar de un tipo anónimo.

Como los métodos Equals y GetHashCode de tipos anónimos se definen en términos de los métodos `Equals` y `GetHashCode` de las propiedades, dos instancias del mismo tipo anónimo son iguales solo si todas sus propiedades son iguales.

Vea también

- Guía de programación de C#
- Inicializadores de objeto y colección
- Introducción a LINQ en C#
- LINQ en C#

Procedimiento para devolver subconjuntos de propiedades de elementos en una consulta (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Use un tipo anónimo en una expresión de consulta cuando se cumplan estas dos condiciones:

- Solo quiere algunas de las propiedades de cada elemento de origen.
- No es necesario almacenar los resultados de la consulta fuera del ámbito del método en el que se ejecuta la consulta.

Si solo quiere devolver una propiedad o campo de cada elemento de origen, puede usar simplemente el operador de punto en la cláusula `select`. Por ejemplo, para devolver solo el `ID` de cada `student`, escriba la cláusula `select` como sigue:

```
select student.ID;
```

Ejemplo

En el ejemplo siguiente se muestra cómo usar un tipo anónimo para devolver solo un subconjunto de las propiedades de cada elemento de origen que coincida con la condición especificada.

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Tenga en cuenta que, si no se especifica ningún nombre, el tipo anónimo usa los nombres del elemento de origen para sus propiedades. Para asignar nombres nuevos a las propiedades del tipo anónimo, escriba la instrucción `select` como sigue:

```
select new { First = student.FirstName, Last = student.LastName };
```

Si lo intenta en el ejemplo anterior, también debe cambiar la instrucción `Console.WriteLine`:

```
Console.WriteLine(student.First + " " + student.Last);
```

Compilar el código

Para ejecutar este código, copie y pegue la clase en una aplicación de consola de C# con una directiva `using` de System.Linq.

Vea también

- Guía de programación de C#
- Tipos anónimos
- LINQ en C#

Interfaces (Guía de programación de C#)

09/04/2020 • 8 minutes to read • Edit Online

Una interfaz contiene las definiciones de un grupo de funcionalidades relacionadas que una clase o una estructura no abstracta deben implementar. Una interfaz puede definir métodos `static`, que deben tener una implementación. A partir de C# 8.0, una interfaz puede definir una implementación predeterminada de miembros. Una interfaz no puede declarar datos de instancia, como campos, propiedades implementadas automáticamente o eventos similares a las propiedades.

Mediante las interfaces puede incluir, por ejemplo, un comportamiento de varios orígenes en una clase. Esta capacidad es importante en C# porque el lenguaje no admite la herencia múltiple de clases. Además, debe usar una interfaz si desea simular la herencia de estructuras, porque no pueden heredar de otra estructura o clase.

Para definir una interfaz, deberá usar la palabra clave interface, tal y como se muestra en el ejemplo siguiente.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

El nombre de una interfaz debe ser un nombre de identificador de C# válido. Por convención, los nombres de interfaz comienzan con una letra `I` mayúscula.

Cualquier clase o estructura que implementa la interfaz IEquatable<T> debe contener una definición para un método Equals que coincida con la firma que la interfaz especifica. Como resultado, puede contar con una clase que implementa `IEquatable<T>` para contener un método `Equals` con el que una instancia de la clase puede determinar si es igual a otra instancia de la misma clase.

La definición de `IEquatable<T>` no facilita ninguna implementación para `Equals`. Una clase o estructura puede implementar varias interfaces, pero una clase solo puede heredar de una sola clase.

Para obtener más información sobre las clases abstractas, vea Clases y miembros de clase abstractos y sellados (Guía de programación de C#).

Las interfaces pueden contener propiedades, eventos, indizadores o métodos de instancia, o bien cualquier combinación de estos cuatro tipos de miembros. Las interfaces pueden contener constructores estáticos, campos, constantes u operadores. Para obtener vínculos a ejemplos, vea Secciones relacionadas. Una interfaz no puede contener campos de instancia, constructores de instancias ni finalizadores. Los miembros de la interfaz son públicos de forma predeterminada.

Para implementar un miembro de interfaz, el miembro correspondiente de la clase de implementación debe ser público, no estático y tener el mismo nombre y firma que el miembro de interfaz.

Cuando una clase o estructura implementa una interfaz, la clase o estructura debe proporcionar una implementación para todos los miembros que la interfaz declara sin proporcionar ninguna implementación predeterminada para ellos. Sin embargo, si una clase base implementa una interfaz, cualquier clase que se derive de la clase base hereda esta implementación.

En el siguiente ejemplo se muestra una implementación de la interfaz IEquatable<T>. La clase de implementación `Car` debe proporcionar una implementación del método Equals.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return (this.Make, this.Model, this.Year) ==
            (car.Make, car.Model, car.Year);
    }
}
```

Las propiedades y los indizadores de una clase pueden definir descriptores de acceso adicionales para una propiedad o indizador que estén definidos en una interfaz. Por ejemplo, una interfaz puede declarar una propiedad que tenga un descriptor de acceso get. La clase que implementa la interfaz puede declarar la misma propiedad con un descriptor de acceso `get` y set. Sin embargo, si la propiedad o el indizador usan una implementación explícita, los descriptores de acceso deben coincidir. Para obtener más información sobre la implementación explícita, vea Implementación de interfaz explícita y Propiedades de interfaces.

Las interfaces pueden heredar de una o varias interfaces. La interfaz derivada hereda los miembros de sus interfaces base. Una clase que implementa una interfaz derivada debe implementar todos los miembros de esta, incluidos los de las interfaces base. Esa clase puede convertirse implícitamente en la interfaz derivada o en cualquiera de sus interfaces base. Una clase puede incluir una interfaz varias veces mediante las clases base que hereda o mediante las interfaces que otras interfaces heredan. Sin embargo, la clase puede proporcionar una implementación de una interfaz solo una vez y solo si la clase declara la interfaz como parte de la definición de la clase (`class ClassName : InterfaceName`). Si la interfaz se hereda porque se heredó una clase base que implementa la interfaz, la clase base proporciona la implementación de los miembros de la interfaz. Sin embargo, la clase derivada puede volver a implementar cualquier miembro de la interfaz virtual, en lugar de usar la implementación heredada. Cuando una interfaz declara una implementación predeterminada de un método, cualquier clase que implemente la interfaz en cuestión hereda esa implementación. Las implementaciones definidas en interfaces son virtuales y es posible que la clase que realice la implementación las invalide.

Una clase base también puede implementar miembros de interfaz mediante el uso de los miembros virtuales. En ese caso, una clase derivada puede cambiar el comportamiento de la interfaz reemplazando los miembros virtuales. Para obtener más información sobre los miembros virtuales, vea Polimorfismo.

Resumen de interfaces

Una interfaz tiene las propiedades siguientes:

- Normalmente, una interfaz es como una clase base abstracta con solo miembros abstractos. Cualquier clase o estructura que implementa la interfaz debe implementar todos sus miembros. Opcionalmente, una interfaz puede definir implementaciones predeterminadas para algunos o todos sus miembros.
- No se puede crear una instancia de una interfaz directamente. Sus miembros se implementan por medio de cualquier clase o estructura que implementa la interfaz.
- Una clase o estructura puede implementar varias interfaces. Una clase puede heredar una clase base y también implementar una o varias interfaces.

Secciones relacionadas

- Propiedades de interfaz
- Indizadores en Interfaces

- Procedimiento para implementar eventos de interfaz
- Clases y structs
- Herencia
- Métodos
- Polimorfismo
- Clases y miembros de clase abstractos y sellados
- Propiedades
- Eventos
- Indizadores

Vea también

- Guía de programación de C#
- Herencia
- Nombres de identificador

Implementación de interfaz explícita (Guía de programación de C#)

19/03/2020 • 4 minutes to read • Edit Online

Si una clase implementa dos interfaces que contienen un miembro con la misma firma, entonces al implementar ese miembro en la clase ambas interfaces usarán ese miembro como su implementación. En el ejemplo siguiente, todas las llamadas a `Paint` invocan el mismo método. En este primer ejemplo se definen los tipos:

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

En el ejemplo siguiente se llama a los métodos:

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();
// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

Cuando dos miembros de interfaz no realizan la misma función, puede provocar una implementación incorrecta de una o ambas interfaces. Es posible implementar un miembro de interfaz de manera explícita: mediante la creación de un miembro de clase que solo se llama a través de la interfaz, y es específico de esa interfaz. Asigne al miembro de clase el nombre de la interfaz y un punto. Por ejemplo:

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

El miembro de clase `IControl.Paint` solo está disponible a través de la interfaz `IControl`, y `ISurface.Paint` solo está disponible mediante `ISurface`. Las dos implementaciones de método son independientes, y ninguna está disponible directamente en la clase. Por ejemplo:

```
// Call the Paint methods from Main.

SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.

IControl c = obj;
c.Paint();  // Calls IControl.Paint on SampleClass.

ISurface s = obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

La implementación explícita también se usa para resolver casos donde dos interfaces declaran miembros diferentes del mismo nombre como una propiedad y un método. Para implementar ambas interfaces, una clase tiene que usar la implementación explícita para la propiedad `P` o el método `P`, o ambos, para evitar un error del compilador. Por ejemplo:

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

A partir de C# 8.0, se puede definir una implementación para los miembros declarados en una interfaz. Si una clase hereda una implementación de método de una interfaz, ese método solo es accesible a través de una referencia del tipo de interfaz. El miembro heredado no aparece como parte de la interfaz pública. En el ejemplo siguiente se define una implementación predeterminada para un método de interfaz:

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

En el ejemplo siguiente se invoca la implementación predeterminada:

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Cualquier clase que implemente la interfaz `IControl` puede invalidar el método `Paint` predeterminado, ya sea como un método público, o bien como una implementación de interfaz explícita.

Vea también

- Guía de programación de C#
- Clases y structs
- Interfaces
- Herencia

Procedimiento Implementar explícitamente miembros de interfaz (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Este ejemplo declara una interfaz, `IDimensions`, y una clase, `Box`, que implementa explícitamente los miembros de interfaz `GetLength` y `GetWidth`. Se tiene acceso a los miembros mediante la instancia de interfaz `dimensions`.

Ejemplo

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Programación sólida

- Tenga en cuenta que las siguientes líneas, en el método `Main`, se comentan porque producirían errores de compilación. No se puede tener acceso a un miembro de interfaz que se implementa explícitamente desde una instancia class:

  ```
  //System.Console.WriteLine("Length: {0}", box1.GetLength());
  //System.Console.WriteLine("Width: {0}", box1.GetWidth());
  ```

- Tenga en cuenta también que las líneas siguientes, en el método `Main`, imprimen correctamente las dimensiones del cuadro porque se llama a los métodos desde una instancia de la interfaz:

```
System.Console.WriteLine("Length: {0}", dimensions.GetLength());
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
```

Vea también

- Guía de programación de C#
- Clases y structs
- Interfaces
- Procedimiento para implementar miembros de dos interfaces de forma explícita

Procedimiento Implementar explícitamente miembros de dos interfaces (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

La implementación explícita de interfaces también permite al programador implementar dos interfaces que tienen los mismos nombres de miembros y dar a cada miembro de una interfaz una implementación independiente. En este ejemplo se muestran las dimensiones de un cuadro en unidades métricas e inglesas. La clase Box implementa dos interfaces, IEnglishDimensions e IMetricDimensions, que representan los diferentes sistemas de medida. Ambas interfaces tienen nombres de miembros idénticos, Length y Width.

Ejemplo

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;

    public Box(float lengthInches, float widthInches)
    {
        this.lengthInches = lengthInches;
        this.widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => lengthInches;

    float IEnglishDimensions.Width() => widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => lengthInches * 2.54f;

    float IMetricDimensions.Width() => widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
        System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

        // Print dimensions in metric units:
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
        System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Programación sólida

Si quiere realizar las medidas en unidades inglesas de manera predeterminada, implemente los métodos Length y Width con normalidad e implemente explícitamente los métodos Length y Width de la interfaz IMetricDimensions:

```
// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;
```

En este caso, se puede tener acceso a las unidades inglesas desde la instancia de clase y acceso a las unidades métricas desde la instancia de interfaz:

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

Vea también

- Guía de programación de C#
- Clases y structs
- Interfaces
- Procedimiento para implementar miembros de interfaz de forma explícita

Delegados (Guía de programación de C#)

18/03/2020 • 5 minutes to read • Edit Online

Un delegado es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Cuando se crea una instancia de un delegado, puede asociar su instancia a cualquier método mediante una signatura compatible y un tipo de valor devuelto. Puede invocar (o llamar) al método a través de la instancia del delegado.

Los delegados se utilizan para pasar métodos como argumentos a otros métodos. Los controladores de eventos no son más que métodos que se invocan a través de delegados. Cree un método personalizado y una clase, como un control de Windows, podrá llamar al método cuando se produzca un determinado evento. En el siguiente ejemplo se muestra una declaración de delegado:

```
public delegate int PerformCalculation(int x, int y);
```

Cualquier método de cualquier clase o struct accesible que coincida con el tipo de delegado se puede asignar al delegado. El método puede ser estático o de instancia. Esto permite cambiar las llamadas a métodos mediante programación y agregar nuevo código a las clases existentes.

> **NOTE**
>
> En el contexto de la sobrecarga de métodos, la signatura de un método no incluye el valor devuelto. Sin embargo, en el contexto de los delegados, la signatura sí lo incluye. En otras palabras, un método debe tener el mismo tipo de valor devuelto que el delegado.

Esta capacidad de hacer referencia a un método como parámetro hace que los delegados sean idóneos para definir métodos de devolución de llamada. Por ejemplo, una referencia a un método que compara dos objetos podría pasarse como argumento a un algoritmo de ordenación. Dado que el código de comparación está en un procedimiento independiente, el algoritmo de ordenación se puede escribir de manera más general.

Información general sobre los delegados

Los delegados tienen las propiedades siguientes:

- Los delegados son similares a los punteros de función de C++, pero los primeros están completamente orientados a objetos y, a diferencia de los punteros de C++ de funciones de miembro, los delegados encapsulan una instancia de objeto y un método.
- Los delegados permiten pasar los métodos como parámetros.
- Los delegados pueden usarse para definir métodos de devolución de llamada.
- Los delegados pueden encadenarse entre sí; por ejemplo, se puede llamar a varios métodos en un solo evento.
- No es necesario que los métodos coincidan exactamente con el tipo de delegado. Para obtener más información, consulte Usar varianza en delegados.
- La versión 2.0 de C# presentó el concepto de los métodos anónimos, los cuales permiten pasar bloques de código como parámetros en lugar de un método definido por separado. En C# 3.0 se presentaron las expresiones lambda como una manera más concisa de escribir bloques de código alineado. Tanto

los métodos anónimos como las expresiones lambda (en ciertos contextos) se compilan en tipos de delegado. Juntas, estas características se conocen ahora como funciones anónimas. Para más información sobre las expresiones lambda, consulte Expresiones lambda.

En esta sección

- Utilizar delegados
- Cuándo usar delegados en lugar de interfaces (Guía de programación de C#)
- Delegados con métodos con nombre y delegados con métodos anónimos
- Uso de varianza en delegados
- Procedimiento para combinar delegados (delegados de multidifusión)
- Procedimiento para declarar un delegado, crear instancias del mismo y usarlo

Especificación del lenguaje C#

Para obtener más información, vea la sección Delegados de Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Capítulos destacados del libro

Delegates, Events, and Lambda Expressions (Delegados, eventos y expresiones lambda) en C# 3.0 Cookbook, Tercera edición: More than 250 solutions for C# 3.0 programmers (Más de 250 soluciones para programadores de C# 3.0)

Delegates and Events (Delegados y eventos) en Learning C# 3.0: Master the fundamentals of C# 3.0

Vea también

- Delegate
- Guía de programación de C#
- Eventos

Utilizar delegados (Guía de programación de C#)

29/06/2020 • 10 minutes to read • Edit Online

Un delegado es un tipo que encapsula de forma segura un método, similar a un puntero de función en C y C++. A diferencia de los punteros de función de C, los delegados están orientados a objetos, proporcionan seguridad de tipos y son seguros. El tipo de un delegado se define por el nombre del delegado. En el ejemplo siguiente, se declara un delegado denominado `Del` que puede encapsular un método que toma una string como argumento y devuelve void:

```
public delegate void Del(string message);
```

Normalmente, un objeto delegado se construye al proporcionar el nombre del método que el delegado encapsulará o con una función anónima. Una vez que se crea una instancia de delegado, el delegado pasará al método una llamada de método realizada al delegado. Los parámetros pasados al delegado por el autor de la llamada se pasan a su vez al método, y el valor devuelto desde el método, si lo hubiera, es devuelto por el delegado al autor de la llamada. Esto se conoce como invocar al delegado. Un delegado con instancias se puede invocar como si fuera el propio método encapsulado. Por ejemplo:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Los tipos de delegado se derivan de la clase Delegate en .NET. Los tipos de delegados son sealed (no se pueden derivar) y no se pueden derivar clases personalizadas de Delegate. Dado que el delegado con instancias es un objeto, puede pasarse como parámetro o asignarse a una propiedad. De este modo, un método puede aceptar un delegado como parámetro y llamar al delegado en algún momento posterior. Esto se conoce como devolución de llamada asincrónica y es un método común para notificar a un llamador que un proceso largo ha finalizado. Cuando se utiliza un delegado de esta manera, el código que usa al delegado no necesita ningún conocimiento de la implementación del método empleado. La funcionalidad es similar a la encapsulación que proporcionan las interfaces.

Otro uso común de devoluciones de llamada es definir un método de comparación personalizado y pasar ese delegado a un método de ordenación. Permite que el código del llamador se convierta en parte del algoritmo de ordenación. En el siguiente método de ejemplo se usa el tipo `Del` como parámetro:

```
public static void MethodWithCallback(int param1, int param2, Del callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

Luego puede pasar el delegado creado anteriormente a ese método:

```
MethodWithCallback(1, 2, handler);
```

Y recibir los siguientes resultados en la consola:

```
The number is: 3
```

Si se usa el delegado como abstracción, no es necesario que `MethodWithCallback` llame directamente a la consola —no tiene que estar diseñado pensando en una consola—. Lo que `MethodWithCallback` hace es simplemente preparar una cadena y pasarla a otro método. Esto es especialmente eficaz, puesto que un método delegado puede utilizar cualquier número de parámetros.

Cuando se crea un delegado para encapsular un método de instancia, el delegado hace referencia tanto a la instancia como al método. Un delegado no tiene conocimiento del tipo de instancia —aparte del método al que encapsula—, por lo que un delegado puede hacer referencia a cualquier tipo de objeto siempre que haya un método en ese objeto que coincida con la signatura del delegado. Cuando se crea un delegado para encapsular un método estático, solo hace referencia al método. Considere las siguientes declaraciones:

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

Junto con el `DelegateMethod` estático mostrado anteriormente, ahora tenemos tres métodos que se pueden encapsularse mediante una instancia de `Del`.

Un delegado puede llamar a más de un método cuando se invoca. Esto se conoce como multidifusión. Para agregar un método adicional a la lista de métodos del delegado —la lista de invocación—, simplemente es necesario agregar dos delegados mediante los operadores de adición o asignación y suma ('+' o '+='). Por ejemplo:

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

En este momento `allMethodsDelegate` contiene tres métodos en su lista de invocación: `Method1`, `Method2` y `DelegateMethod`. Los tres delegados originales, `d1`, `d2` y `d3`, permanecen sin cambios. Cuando se invoca `allMethodsDelegate`, todos los métodos se llaman en orden. Si el delegado usa parámetros de referencia, la referencia se pasa secuencialmente a cada uno de los tres métodos por turnos, y cualquier cambio que realice un método es visible para el siguiente método. Cuando alguno de los métodos produce una excepción que no se captura dentro del método, esa excepción se pasa al llamador del delegado y no se llama a los métodos siguientes de la lista de invocación. Si el delegado tiene un valor devuelto o los parámetros de salida, devuelve el valor devuelto y los parámetros del último método invocado. Para quitar un método de la lista de invocación, utilice los operadores de decremento o de asignación de decremento (`-` o `-=`). Por ejemplo:

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Dado que los tipos de delegado se derivan de `System.Delegate`, los métodos y propiedades definidas por esa clase se pueden llamar en el delegado. Por ejemplo, para buscar el número de métodos en la lista de invocación de un delegado, se puede escribir:

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Los delegados con más de un método en su lista de invocación derivan de MulticastDelegate, que es una subclase de `System.Delegate`. El código anterior funciona en ambos casos porque las dos clases admiten `GetInvocationList`.

Los delegados de multidifusión se utilizan mucho en el control de eventos. Los objetos de origen de evento envían notificaciones de evento a los objetos de destinatario que se han registrado para recibir ese evento. Para suscribirse a un evento, el destinatario crea un método diseñado para controlar el evento; a continuación, crea a un delegado para dicho método y pasa el delegado al origen de eventos. El origen llama al delegado cuando se produce el evento. Después, el delegado llama al método que controla los eventos en el destinatario y entrega los datos del evento. El origen del evento define el tipo de delegado para un evento determinado. Para obtener más información, consulte Eventos.

La comparación de delegados de dos tipos distintos asignados en tiempo de compilación generará un error de compilación. Si las instancias de delegado son estáticamente del tipo `System.Delegate`, entonces se permite la comparación, pero devolverá false en tiempo de ejecución. Por ejemplo:

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

Vea también

- Guía de programación de C#
- Delegados
- Uso de varianza en delegados
- Varianza en delegados
- Uso de varianza para los delegados genéricos Func y Action
- Eventos

Delegados con métodos con nombre y Métodos anónimos (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

Un delegado puede asociarse con un método con nombre. Cuando crea una instancia de un delegado mediante un método con nombre, el método se pasa como un parámetro, por ejemplo:

```
// Declare a delegate.
delegate void Del(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
Del d = obj.DoWork;
```

Esto se llama con un método con nombre. Los delegados construidos con un método con nombre pueden encapsular un método estático o un método de instancia. Los métodos con nombre son la única manera de crear una instancia de un delegado en versiones anteriores de C#. En cambio, en una situación en la que crear un método nuevo es una sobrecarga no deseada, C# le permite crear una instancia de un delegado y especificar inmediatamente un bloque de código que el delegado procesará cuando se llame. El bloque puede contener una expresión lambda o un método anónimo. Para obtener más información, vea Funciones anónimas.

Comentarios

El método que pasa como un parámetro de delegado debe tener la misma firma que la declaración de delegado.

Una instancia de delegado puede encapsular un método de instancia o estático.

Aunque el delegado puede usar un parámetro out, no recomendamos su uso con delegados de eventos de multidifusión porque no puede conocer a qué delegado se llamará.

Ejemplo 1

A continuación se muestra un ejemplo sencillo de cómo declarar y usar un delegado. Tenga en cuenta que tanto el delegado, `Del`, como el método asociado, `MultiplyNumbers`, tienen la misma firma

```
// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
    2 4 6 8 10
*/
```

Ejemplo 2

En el ejemplo siguiente, un delegado se asigna a métodos estáticos y de instancia y devuelve información específica de cada uno.

```
// Declare a delegate
delegate void Del();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Del d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

Vea también

- Guía de programación de C#
- Delegados
- Procedimiento para combinar delegados (delegados de multidifusión)
- Eventos

Procedimiento para combinar delegados (delegados de multidifusión) (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

En este ejemplo se muestra cómo crear delegados de multidifusión. Una propiedad útil de los objetos delegados es que puedan asignarse objetos múltiples a una instancia de delegado con el operador `+`. El delegado de multidifusión contiene una lista de los delegados asignados. Cuando se llama al delegado de multidifusión, invoca a los delegados de la lista, en orden. Solo los delegados del mismo tipo pueden combinarse.

El operador `-` puede usarse para quitar un delegado de componente de un delegado de multidifusión.

Ejemplo

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

Vea también

- MulticastDelegate
- Guía de programación de C#
- Eventos

Procedimiento Declarar un delegado, crear instancias del mismo y utilizarlo (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

En C# 1.0 y versiones posteriores, los delegados se pueden declarar como se muestra en el ejemplo siguiente.

```
// Declare a delegate.
delegate void Del(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

```
// Create an instance of the delegate.
Del del1 = new Del(Notify);
```

C# 2.0 ofrece una forma más sencilla de escribir la declaración anterior, tal como se muestra en el ejemplo siguiente.

```
// C# 2.0 provides a simpler way to declare an instance of Del.
Del del2 = Notify;
```

En C# 2.0 y versiones posteriores, también es posible utilizar un método anónimo para declarar e inicializar un delegado, como se muestra en el ejemplo siguiente.

```
// Instantiate Del by using an anonymous method.
Del del3 = delegate(string name)
    { Console.WriteLine($"Notification received for: {name}"); };
```

En C# 3.0 y versiones posteriores, también se pueden declarar los delegados y crear una instancia de ellos mediante una expresión lambda, como se muestra en el ejemplo siguiente.

```
// Instantiate Del by using a lambda expression.
Del del4 = name =>  { Console.WriteLine($"Notification received for: {name}"); };
```

Para obtener más información, vea Expresiones lambda.

En el ejemplo siguiente se ilustra cómo declarar un delegado, crear una instancia del mismo y utilizarlo La clase `BookDB` encapsula una base de datos de una librería que mantiene una base de datos de libros. Expone un método `ProcessPaperbackBooks`, que busca todos los libros de bolsillo en la base de datos y llama a un delegado para cada uno. El tipo `delegate` utilizado se denomina `ProcessBookDelegate`. La clase `Test` utiliza esta clase para imprimir los títulos y el precio medio de los libros de bolsillo.

El uso de delegados promueve una separación adecuada de la funcionalidad entre la base de datos de la librería y el código de cliente. El código de cliente no sabe cómo se almacenan los libros ni cómo el código de la biblioteca encuentra los libros de bolsillo. El código de la librería no sabe qué procesamiento se realiza con los libros de bolsillo después de que los encuentra.

Ejemplo

```
// A set of classes for handling a bookstore:
namespace Bookstore
{
    using System.Collections;

    // Describes a book in the book list:
    public struct Book
    {
        public string Title;        // Title of the book.
        public string Author;       // Author of the book.
        public decimal Price;       // Price of the book.
        public bool Paperback;      // Is it paperback?

        public Book(string title, string author, decimal price, bool paperBack)
        {
            Title = title;
            Author = author;
            Price = price;
            Paperback = paperBack;
        }
    }

    // Declare a delegate type for processing a book:
    public delegate void ProcessBookDelegate(Book book);

    // Maintains a book database.
    public class BookDB
    {
        // List of all books in the database:
        ArrayList list = new ArrayList();

        // Add a book to the database:
        public void AddBook(string title, string author, decimal price, bool paperBack)
        {
            list.Add(new Book(title, author, price, paperBack));
        }

        // Call a passed-in delegate on each paperback book to process it:
        public void ProcessPaperbackBooks(ProcessBookDelegate processBook)
        {
            foreach (Book b in list)
            {
                if (b.Paperback)
                    // Calling the delegate:
                    processBook(b);
            }
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
```

```
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
        {
            Console.WriteLine($"   {b.Title}");
        }

        // Execution starts here.
        static void Main()
        {
            BookDB bookDB = new BookDB();

            // Initialize the database with some books:
            AddBooks(bookDB);

            // Print all the titles of paperbacks:
            Console.WriteLine("Paperback Book Titles:");

            // Create a new delegate object associated with the static
            // method Test.PrintTitle:
            bookDB.ProcessPaperbackBooks(PrintTitle);

            // Get the average price of a paperback by using
            // a PriceTotaller object:
            PriceTotaller totaller = new PriceTotaller();

            // Create a new delegate object associated with the nonstatic
            // method AddBookToTotal on the object totaller:
            bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

            Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
                    totaller.AveragePrice());
        }

        // Initialize the book database with some test books:
        static void AddBooks(BookDB bookDB)
        {
            bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and Dennis M. Ritchie", 19.95m,
true);
            bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium", 39.95m, true);
            bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m, false);
            bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams", 12.00m, true);
        }
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

Programación sólida

- Declaración de un delegado.

La instrucción siguiente declara un nuevo tipo de delegado.

```
public delegate void ProcessBookDelegate(Book book);
```

Cada tipo de delegado describe el número y los tipos de argumentos, y el tipo del valor devuelto de los métodos que puede encapsular. Siempre que se necesite un nuevo conjunto de tipos de argumentos o de tipos de valores devueltos, se debe declarar un nuevo tipo de delegado.

- Creación de instancias de un delegado.

  Después de haber declarado un tipo de delegado, debe crearse un objeto delegado y asociarse con un método particular. En el ejemplo anterior, esto se hace pasando el método `PrintTitle` al método `ProcessPaperbackBooks` como en el ejemplo siguiente:

```
bookDB.ProcessPaperbackBooks(PrintTitle);
```

  Esto crea un objeto delegado nuevo asociado con el método estático `Test.PrintTitle`. De forma similar, el método no estático `AddBookToTotal` del objeto `totaller` se pasa como en el ejemplo siguiente:

```
bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
```

  En ambos casos, se pasa un nuevo objeto delegado al método `ProcessPaperbackBooks`.

  Después de que se ha creado un delegado, el método con el que está asociado nunca cambia; los objetos delegados son inmutables.

- Llamada a un delegado.

  Después de haber creado un objeto delegado, este suele pasarse a otro código que llamará al delegado. La llamada a un objeto delegado se realiza mediante la utilización del nombre de dicho objeto, seguido de los argumentos entre paréntesis que se deben pasar al delegado. A continuación se expone un ejemplo de llamada a un delegado:

```
processBook(b);
```

  Se puede llamar a un delegado de forma sincrónica, como en este ejemplo, o bien de forma asincrónica con los métodos `BeginInvoke` y `EndInvoke`.

Vea también

- Guía de programación de C#
- Eventos
- Delegados

Matrices (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

Puede almacenar varias variables del mismo tipo en una estructura de datos de matriz. Puede declarar una matriz mediante la especificación del tipo de sus elementos. Si quiere que la matriz almacene elementos de cualquier tipo, puede especificar `object` como su tipo. En el sistema de tipos unificado de C#, todos los tipos, los predefinidos y los definidos por el usuario, los tipos de referencia y los tipos de valores, heredan directa o indirectamente de Object.

```
type[] arrayName;
```

Ejemplo

Los ejemplos siguientes crean matrices unidimensionales, multidimensionales y escalonadas:

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Información general sobre la matriz

Una matriz tiene las propiedades siguientes:

- Puede ser una matriz unidimensional, multidimensional o escalonada.
- El número de dimensiones y la longitud de cada dimensión se establecen al crear la instancia de matriz. No se pueden cambiar estos valores durante la vigencia de la instancia.
- Los valores predeterminados de los elementos numéricos de matriz se establecen en cero y los elementos de referencia se establecen en null.
- Una matriz escalonada es una matriz de matrices y, por consiguiente, sus elementos son tipos de referencia y se inicializan en `null`.

- Las matrices se indexan con cero: una matriz con `n` elementos se indexa de `0` a `n-1`.
- Los elementos de una matriz puede ser cualquier tipo, incluido un tipo de matriz.
- Los tipos de matriz son tipos de referencia que proceden del tipo base abstracto Array. Como este tipo implementa IEnumerable y IEnumerable<T>, puede usar la iteración foreach en todas las matrices de C#.

Secciones relacionadas

- Matrices como objetos
- Utilizar foreach con matrices
- Pasar matrices como argumentos

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Colecciones

Utilizar matrices como objetos (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

En C#, las matrices son actualmente objetos, y no simplemente regiones direccionables de memoria contigua como en C y C++. Array es el tipo base abstracto de todos los tipos de matriz. Puede usar las propiedades, y otros miembros de clase, que tiene Array. Un ejemplo de esto sería usar la propiedad Length para obtener la longitud de una matriz. El código siguiente asigna la longitud de la matriz `numbers`, que es `5`, a una variable denominada `lengthOfNumbers`:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

La clase Array proporciona muchos otros métodos útiles y propiedades para ordenar, buscar y copiar matrices.

Ejemplo

Este ejemplo usa la propiedad Rank para mostrar el número de dimensiones de una matriz.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

Vea también

- Guía de programación de C#
- Matrices
- Matrices unidimensionales
- Matrices multidimensionales
- Matrices escalonadas

Matrices unidimensionales (Guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

Cree una matriz unidimensional mediante el operador new; para ello, especifique el tipo de elemento de matriz y el número de elementos. En el ejemplo siguiente se declara una matriz de cinco enteros:

```
int[] array = new int[5];
```

Esta matriz contiene los elementos de `array[0]` a `array[4]`. Los elementos de la matriz se inicializan en el valor predeterminado del tipo de elemento, `0` para los enteros.

Las matrices pueden almacenar cualquier tipo de elemento que se especifique, como en el ejemplo siguiente, en el que se declara una matriz de cadenas:

```
string[] stringArray = new string[6];
```

Inicialización de matriz

Puede inicializar los elementos de una matriz al declararla. El especificador de longitud no es necesario porque la medida se infiere a partir del número de elementos de la lista de inicialización. Por ejemplo:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

En el código siguiente se muestra una declaración de una matriz de cadenas donde cada elemento de la matriz se inicializa mediante el nombre de un día:

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Puede evitar la expresión `new` y el tipo de matriz al inicializar una matriz en la declaración, tal como se muestra en el código siguiente. Esto se conoce como matriz con tipo implícito:

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Puede declarar una variable de matriz sin crearla, pero debe usar el operador `new` cuando asigne una nueva matriz a esta variable. Por ejemplo:

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK
//array3 = {1, 3, 5, 7, 9};   // Error
```

Matrices de tipo de valor y tipo de referencia

Tenga en cuenta la siguiente declaración de matriz:

```
SomeType[] array4 = new SomeType[10];
```

El resultado de esta instrucción depende de si `SomeType` es un tipo de valor o un tipo de referencia. Si es un tipo de valor, la instrucción crea una matriz de 10 elementos, y cada uno de ellos tiene el tipo `SomeType`. Si `SomeType` es un tipo de referencia, la instrucción crea una matriz de 10 elementos y cada uno de ellos se inicializa en una referencia nula. En ambas instancias, los elementos se inicializan en el valor predeterminado para el tipo de elemento. Para obtener más información sobre los tipos de valor y de referencia, consulte Tipos de valor y Tipos de referencia.

Vea también

- Array
- Matrices
- Matrices multidimensionales
- Matrices escalonadas

Matrices multidimensionales (Guía de programación de C#)

04/05/2020 • 3 minutes to read • Edit Online

Las matrices pueden tener varias dimensiones. Por ejemplo, la siguiente declaración crea una matriz bidimensional de cuatro filas y dos columnas.

```
int[,] array = new int[4, 2];
```

La siguiente declaración crea una matriz de tres dimensiones, 4, 2 y 3.

```
int[, ,] array1 = new int[4, 2, 3];
```

Inicialización de matriz

La matriz se puede inicializar en la declaración como se muestra en el ejemplo siguiente.

```
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                        { "five", "six" } };

// Three-dimensional array.
int[, ,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                 { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[, ,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                       { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++) {
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```

También se puede inicializar la matriz sin especificar el rango.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Si opta por declarar una variable de matriz sin inicializarla, deberá usar el operador `new` para asignar una matriz a la variable. El uso de `new` se muestra en el ejemplo siguiente.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };   // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

En el ejemplo siguiente se asigna un valor a un elemento de matriz determinado.

```
array5[2, 1] = 25;
```

De igual forma, en el ejemplo siguiente se obtiene el valor de un elemento de matriz determinado y se asigna a la

variable `elementValue`.

```
int elementValue = array5[2, 1];
```

El ejemplo de código siguiente inicializa los elementos de matriz con valores predeterminados (salvo las matrices escalonadas).

```
int[,] array6 = new int[10, 10];
```

Vea también

- Guía de programación de C#
- Matrices
- Matrices unidimensionales
- Matrices escalonadas

Matrices escalonadas (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Una matriz escalonada es una matriz cuyos elementos son matrices. Los elementos de una matriz escalonada pueden ser de diferentes dimensiones y tamaños. Una matriz escalonada se denomina a veces "matriz de matrices". En los ejemplos siguientes, se muestra cómo declarar, inicializar y acceder a matrices escalonadas.

La siguiente es una declaración de una matriz unidimensional que tiene tres elementos, cada uno de los cuales es una matriz unidimensional de enteros:

```
int[][] jaggedArray = new int[3][];
```

Para poder usar `jaggedArray`, se deben inicializar sus elementos. Puede inicializar los elementos de esta forma:

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

Cada uno de los elementos es una matriz unidimensional de enteros. El primer elemento es una matriz de 5 enteros, el segundo es una matriz de 4 enteros y el tercero es una matriz de 2 enteros.

También es posible usar inicializadores para rellenar los elementos de matriz con valores, en cuyo caso no es necesario el tamaño de la matriz. Por ejemplo:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

También puede inicializar la matriz en la declaración de esta forma:

```
int[][] jaggedArray2 = new int[][]
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

Puede usar la siguiente forma abreviada. Tenga en cuenta que no puede omitir el operador `new` de la inicialización de elementos porque no hay ninguna inicialización predeterminada para los elementos:

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

Una matriz escalonada es una matriz de matrices y, por consiguiente, sus elementos son tipos de referencia y se inicializan en `null`.

Puede tener acceso a elementos individuales de la matriz como en estos ejemplos:

```
// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray3[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray3[2][1] = 88;
```

Es posible mezclar matrices multidimensionales y escalonadas. La siguiente es una declaración e inicialización de una matriz escalonada unidimensional que contiene tres elementos de matriz bidimensional de tamaños diferentes. Para obtener más información sobre las matrices bidimensionales, vea Matrices multidimensionales.

```
int[][,] jaggedArray4 = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};
```

Puede tener acceso a los elementos individuales como se muestra en este ejemplo, que muestra el valor del elemento `[1,0]` de la primera matriz (valor `5`):

```
System.Console.Write("{0}", jaggedArray4[0][1, 0]);
```

El método `Length` devuelve el número de matrices contenidos en la matriz escalonada. Por ejemplo, suponiendo que ha declarado la matriz anterior, esta línea:

```
System.Console.WriteLine(jaggedArray4.Length);
```

devuelve un valor de 3.

Ejemplo

En este ejemplo, se crea una matriz cuyos elementos son matrices. Cada uno de los elementos de matriz tiene un tamaño diferente.

```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

        // Initialize the elements.
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements.
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Vea también

- Array
- Guía de programación de C#
- Matrices
- Matrices unidimensionales
- Matrices multidimensionales

Uso de foreach con matrices (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

La instrucción foreach ofrece una manera sencilla y limpia de iterar los elementos de una matriz.

Para matrices unidimensionales, la instrucción `foreach` procesa los elementos en orden creciente de índice, comenzando con el índice 0 y terminando con el índice `Length - 1`:

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

En el caso de matrices multidimensionales, los elementos se recorren de tal manera que primero se incrementan los índices de la dimensión más a la derecha, luego la siguiente dimensión a la izquierda, y así sucesivamente a la izquierda:

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

En cambio, con las matrices multidimensionales, usar un bucle for anidado ofrece un mayor control sobre el orden en el que se procesan los elementos de la matriz.

Vea también

- Array
- Guía de programación de C#
- Matrices
- Matrices unidimensionales
- Matrices multidimensionales
- Matrices escalonadas

Pasar matrices como argumentos (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Las matrices se pueden pasar como argumentos a parámetros de método. Dado que las matrices son tipos de referencia, el método puede cambiar el valor de los elementos.

Pasar matrices unidimensionales como argumentos

Puede pasar una matriz unidimensional inicializada a un método. Por ejemplo, la siguiente instrucción envía una matriz a un método de impresión.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

En el siguiente código, se muestra una implementación parcial del método de impresión.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

Puede inicializar y pasar una nueva matriz en un solo paso, como se muestra en el ejemplo siguiente.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

Ejemplo

En el ejemplo siguiente, una matriz de cadenas se inicializa y pasa como un argumento a un método `DisplayArray` para cadenas. El método muestra los elementos de la matriz. A continuación, el método `ChangeArray` invierte los elementos de la matriz y, después, el método `ChangeArrayElements` modifica los tres primeros elementos de la matriz. Después de cada devolución de método, el método `DisplayArray` muestra que pasar una matriz por valor no impide que se cambien los elementos de la matriz.

```
using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Pasar matrices multidimensionales como argumentos

Pase una matriz multidimensional inicializada a un método de la misma manera que pasa una matriz unidimensional.

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);
```

En el código siguiente, se muestra una declaración parcial de un método de impresión que acepta una matriz bidimensional como su argumento.

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

Puede inicializar y pasar una matriz nueva en un solo paso, como se muestra en el ejemplo siguiente:

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

Ejemplo

En el ejemplo siguiente, una matriz bidimensional de enteros se inicializa y pasa al método `Print2DArray`. El método muestra los elementos de la matriz.

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
    /* Output:
        Element(0,0)=1
        Element(0,1)=2
        Element(1,0)=3
        Element(1,1)=4
        Element(2,0)=5
        Element(2,1)=6
        Element(3,0)=7
        Element(3,1)=8
    */
```

Vea también

- Guía de programación de C#
- Matrices
- Matrices unidimensionales
- Matrices multidimensionales
- Matrices escalonadas

Matrices con asignación implícita de tipos (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Puede crear una matriz con tipo implícito en la que se deduce el tipo de la instancia de matriz de los elementos especificados en el inicializador de matriz. Las reglas de cualquier variable con tipo implícito también se aplican a las matrices con tipo implícito. Para más información, vea Variables locales con asignación implícita de tipos.

Normalmente, se usan matrices con tipo implícito en expresiones de consulta junto con tipos anónimos e inicializadores de objeto y colección.

En los ejemplos siguientes, se muestra cómo crear una matriz con tipo implícito:

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

En el ejemplo anterior observe que, con las matrices con tipo implícito, no se usan corchetes en el lado izquierdo de la instrucción de inicialización. Tenga en cuenta también que las matrices escalonadas se inicializan mediante `new []` al igual que las matrices unidimensionales.

Matrices con tipo implícito en inicializadores de objeto

Al crear un tipo anónimo que contiene una matriz, esta debe tener tipo implícito en el inicializador de objeto del tipo. En el ejemplo siguiente, `contacts` es una matriz con tipos implícitos anónimos, cada uno de los cuales contiene una matriz denominada `PhoneNumbers`. Tenga en cuenta que la palabra clave `var` no se usa dentro de los inicializadores de objeto.

```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

Vea también

- Guía de programación de C#
- Variables locales con asignación implícita de tipos
- Matrices
- Tipos anónimos
- Inicializadores de objeto y colección
- var
- LINQ en C#

Cadenas (Guía de programación de C#)

18/03/2020 • 22 minutes to read • Edit Online

Una cadena es un objeto de tipo String cuyo valor es texto. Internamente, el texto se almacena como una colección secuencial de solo lectura de objetos Char. No hay ningún carácter que finaliza en null al final de una cadena de C#; por lo tanto, la cadena de C# puede contener cualquier número de caracteres nulos insertados ('\0'). La propiedad Length de una cadena representa el número de objetos `Char` que contiene, no el número de caracteres Unicode. Para obtener acceso a los puntos de código Unicode individuales de una cadena, use el objeto StringInfo.

cadena frente System.String

En C#, la palabra clave `string` es un alias de String. Por lo tanto, `String` y `string` son equivalentes y se puede utilizar la convención de nomenclatura que prefiera. La clase `String` proporciona muchos métodos para crear, manipular y comparar cadenas de forma segura. Además, el lenguaje C# sobrecarga algunos operadores para simplificar las operaciones de cadena comunes. Para más información sobre la palabra clave, consulte string. Para obtener más información sobre el tipo y sus métodos, vea String.

Declaración e inicialización de cadenas

Puede declarar e inicializar cadenas de varias maneras, tal como se muestra en el ejemplo siguiente:

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Tenga en cuenta que no se usa el operador new para crear un objeto de cadena, salvo cuando se inicialice la

cadena con una matriz de caracteres.

Inicialice una cadena con el valor constante Empty para crear un objeto String cuya cadena tenga longitud cero. La representación literal de la cadena de una cadena de longitud cero es "". Mediante la inicialización de las cadenas con el valor Empty en lugar de null, puede reducir las posibilidades de que se produzca una excepción NullReferenceException. Use el método estático IsNullOrEmpty(String) para comprobar el valor de una cadena antes de intentar obtener acceso a ella.

Inmutabilidad de los objetos de cadena

Los objetos de cadena son *inmutables*: no se pueden cambiar después de haberse creado. Todos los métodos String y operadores de C# que parecen modificar una cadena en realidad devuelven los resultados en un nuevo objeto de cadena. En el siguiente ejemplo, cuando el contenido de `s1` y `s2` se concatena para formar una sola cadena, las dos cadenas originales no se modifican. El operador `+=` crea una nueva cadena que contiene el contenido combinado. Este nuevo objeto se asigna a la variable `s1` y el objeto original que se asignó a `s1` se libera para la recolección de elementos no utilizados porque ninguna otra variable contiene una referencia a él.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Dado que una "modificación" de cadena es en realidad una creación de cadena, debe tener cuidado al crear referencias a las cadenas. Si crea una referencia a una cadena y después "modifica" la cadena original, la referencia seguirá apuntando al objeto original en lugar de al objeto nuevo creado al modificarse la cadena. El código siguiente muestra este comportamiento:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

Para más información acerca de cómo crear cadenas nuevas basadas en modificaciones como las operaciones de buscar y reemplazar en la cadena original, consulte Modificación del contenido de cadenas.

Literales de cadena regulares y textuales

Utilice literales de cadena regulares cuando tenga que insertar caracteres de escape proporcionados por C#, tal como se muestra en el ejemplo siguiente:

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Utilice cadenas textuales para mayor comodidad y mejor legibilidad cuando el texto de la cadena contenga caracteres de barra diagonal inversa, por ejemplo, en rutas de acceso de archivo. Como las cadenas textuales conservan los caracteres de nueva línea como parte del texto de la cadena, pueden utilizarse para inicializar cadenas multilíneas. Utilice comillas dobles para insertar una comilla simple dentro de una cadena textual. En el ejemplo siguiente se muestran algunos usos habituales de las cadenas textuales:

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

Secuencias de escape de cadena

SECUENCIA DE ESCAPE	NOMBRE DE CARÁCTER	CODIFICACIÓN UNICODE
\'	Comilla simple	0x0027
\"	Comilla doble	0x0022
\\	Barra diagonal inversa	0x005C
\0	Null	0x0000
\a	Alerta	0x0007
\b	Retroceso	0x0008
\f	Avance de página	0x000C
\n	Nueva línea	0x000A
\r	Retorno de carro	0x000D

SECUENCIA DE ESCAPE	NOMBRE DE CARÁCTER	CODIFICACIÓN UNICODE
\t	Tabulación horizontal	0x0009
\v	Tabulación vertical	0x000B
\u	Secuencia de escape Unicode (UTF-16)	`\uHHHH` (intervalo: 0000 - FFFF; ejemplo: `\u00E7` = "ç")
\U	Secuencia de escape Unicode (UTF-32)	`\U00HHHHHH` (intervalo: 000000 - 10FFFF; ejemplo: `\U0001F47D` = "👽")
\x	Secuencia de escape Unicode similar a "\u" excepto con longitud variable	`\xH[H][H][H]` (intervalo: 0 - FFFF; ejemplo: `\x00E7` o `\x0E7` o `\xE7` = "ç")

WARNING

Cuando se usa la secuencia de escape `\x` y se especifican menos de 4 dígitos hexadecimales, si los caracteres que van inmediatamente después de la secuencia de escape son dígitos hexadecimales válidos (es decir, 0-9, A-f y a-f), se interpretará que forman parte de la secuencia de escape. Por ejemplo, `\xA1` genera "¡", que es el punto de código U+00A1. Sin embargo, si el carácter siguiente es "A" o "a", la secuencia de escape se interpretará como `\xA1A` y producirá "■", que es el punto de código U+0A1A. En casos así, se pueden especificar los 4 dígitos hexadecimales (por ejemplo, `\x00A1`) para evitar posibles errores de interpretación.

NOTE

En tiempo de compilación, las cadenas textuales se convierten en cadenas normales con las mismas secuencias de escape. Por lo tanto, si se muestra una cadena textual en la ventana Inspección del depurador, verá los caracteres de escape agregados por el compilador, no la versión textual del código fuente. Por ejemplo, la cadena textual `@"C:\files.txt"` aparecerá en la ventana Inspección, como "C:\\files.txt".

Cadenas de formato

Una cadena de formato es una cadena cuyo contenido se determina de manera dinámica en tiempo de ejecución. Las cadenas de formato se crean mediante la inserción de *expresiones interpoladas* o marcadores de posición entre llaves dentro de una cadena. Todo lo incluido entre llaves (`{...}`) se resolverá en un valor y se generará como una cadena con formato en tiempo de ejecución. Existen dos métodos para crear cadenas de formato: interpolación de cadenas y formato compuesto.

Interpolación de cadenas

Disponible en C# 6.0 y versiones posteriores, las *cadenas interpoladas* se identifican por el carácter especial `$` e incluyen expresiones interpoladas entre llaves. Si no está familiarizado con la interpolación de cadenas, consulte el tutorial interactivo Interpolación de cadenas en C# para obtener información general rápidamente.

Use la interpolación de cadenas para mejorar la legibilidad y el mantenimiento del código. Con la interpolación de cadenas se obtienen los mismos resultados que con el método `String.Format`, pero mejora la facilidad de uso y la claridad en línea.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Formatos compuestos

String.Format emplea marcadores de posición entre llaves para crear una cadena de formato. Los resultados de este ejemplo son similares a la salida del método de interpolación de cadenas usado anteriormente.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

Para más información sobre cómo dar formato a los tipos .NET, consulte Aplicar formato a tipos en .NET.

Subcadenas

Una subcadena es cualquier secuencia de caracteres que se encuentra en una cadena. Use el método Substring para crear una nueva cadena de una parte de la cadena original. Puede buscar una o más apariciones de una subcadena con el método IndexOf. Use el método Replace para reemplazar todas las apariciones de una subcadena especificada por una nueva cadena. Al igual que el método Substring, Replace devuelve una cadena nueva y no modifica la cadena original. Para más información, consulte Cómo: Buscar cadenas y Procedimiento para modificar el contenido de cadenas.

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Acceso a caracteres individuales

Puede utilizar la notación de matriz con un valor de índice para adquirir acceso de solo lectura a caracteres individuales, como en el ejemplo siguiente:

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

Si el método String no proporciona la funcionalidad que debe tener para modificar los caracteres individuales de una cadena, puede usar un objeto StringBuilder para modificar los caracteres individuales "en contexto" y, después, crear una cadena para almacenar los resultados mediante el método StringBuilder. En el ejemplo siguiente, se supone que debe modificar la cadena original de una manera determinada y, después, almacenar los resultados para un uso futuro:

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Cadenas nulas y cadenas vacías

Una cadena vacía es una instancia de un objeto System.String que contiene cero caracteres. Las cadenas vacías se utilizan a menudo en distintos escenarios de programación para representar un campo de texto en blanco. Puede llamar a métodos en cadenas vacías porque son objetos System.String válidos. Las cadenas vacías se inicializan como sigue:

```
string s = String.Empty;
```

En cambio, una cadena nula no hace referencia a una instancia de un objeto System.String y cualquier intento de llamar a un método en una cadena nula produce una excepción NullReferenceException. Sin embargo, puede utilizar cadenas nulas en operaciones de comparación y concatenación con otras cadenas. Los ejemplos siguientes muestran algunos casos en que una referencia a una cadena nula provoca y no provoca una excepción:

```
static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}
```

Uso de StringBuilder para la creación rápida de cadenas

Las operaciones de cadena en .NET están muy optimizadas y en la mayoría de los casos no afectan significativamente al rendimiento. Sin embargo, en algunos escenarios, como los bucles de pequeñas dimensiones que se ejecutan cientos o miles de veces, las operaciones de cadena pueden afectar al rendimiento. La clase StringBuilder crea un búfer de cadena que proporciona un mejor rendimiento si el programa realiza muchas manipulaciones de cadenas. La cadena StringBuilder también permite reasignar caracteres individuales, algo que el tipo de datos de cadena integrado no admite. Por ejemplo, este código cambia el contenido de una cadena sin crear una nueva:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

En este ejemplo, se usa un objeto StringBuilder para crear una cadena a partir de un conjunto de tipos numéricos:

```
using System;
using System.Text;

namespace CSRefStrings
{
    class TestStringBuilder
    {
        static void Main()
        {
            var sb = new StringBuilder();

            // Create a string composed of numbers 0 - 9
            for (int i = 0; i < 10; i++)
            {
                sb.Append(i.ToString());
            }
            Console.WriteLine(sb);  // displays 0123456789

            // Copy one character of the string (not possible with a System.String)
            sb[0] = sb[9];

            Console.WriteLine(sb);  // displays 9123456789
            Console.WriteLine();
        }
    }
}
```

Cadenas, métodos de extensión y LINQ

Dado que el tipo String implementa IEnumerable<T>, puede usar los métodos de extensión definidos en la clase Enumerable en cadenas. Para evitar el desorden visual, estos métodos se excluyen de IntelliSense para el tipo String, pero aun así están disponibles. También puede usar expresiones de consulta LINQ en cadenas. Para más información, consulte LINQ y cadenas.

Temas relacionados

TEMA	DESCRIPCIÓN
Procedimiento para modificar el contenido de cadenas	Muestra técnicas para transformar cadenas y modificar el contenido de estas.
Comparación de cadenas	Muestra cómo realizar comparaciones ordinales y culturales específicas de las cadenas.
Concatenación de varias cadenas	Muestra diferentes maneras de combinar varias cadenas en una.
Análisis de cadenas mediante String.Split	Contiene ejemplos de código que muestran cómo utilizar el método `String.Split` para analizar cadenas.
Cómo: Buscar cadenas	Explica cómo usar la búsqueda para especificar texto o patrones en cadenas.
Determinación de si una cadena representa un valor numérico	Muestra cómo analizar de forma segura una cadena para ver si tiene un valor numérico válido.

TEMA	DESCRIPCIÓN
Interpolación de cadenas	Describe la característica de interpolación de cadena que proporciona una sintaxis adecuada para dar formato a las cadenas.
Operaciones básicas de cadenas	Proporciona vínculos a temas que usan los métodos System.String y System.Text.StringBuilder para realizar operaciones básicas de cadenas.
Analizar cadenas	Describe cómo convertir las representaciones de cadena de los tipos base de .NET en las instancias de los tipos correspondientes.
Analizar cadenas de fecha y hora en .NET	Muestra cómo convertir una cadena como "24/01/2008" en un objeto System.DateTime.
Comparar cadenas	Incluye información sobre cómo comparar cadenas y proporciona ejemplos de C# y Visual Basic.
Utilizar la clase StringBuilder	Describe cómo crear y modificar objetos de cadena dinámicos con la clase StringBuilder.
LINQ y cadenas	Proporciona información sobre cómo realizar varias operaciones de cadena utilizando consultas LINQ.
Guía de programación de C#	Proporciona vínculos a temas que explican las construcciones de programación en C#.

Procedimiento Determinar si una cadena representa un valor numérico (Guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

Para determinar si una cadena es una representación válida de un tipo numérico especificado, use el método estático `TryParse` implementado por todos los tipos numéricos primitivos y también por tipos como DateTime y IPAddress. En el ejemplo siguiente se muestra cómo determinar si "108" es un valor int válido.

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

Si la cadena contiene caracteres no numéricos o el valor numérico es demasiado grande o demasiado pequeño para el tipo determinado que ha especificado, `TryParse` devuelve el valor false y establece el parámetro out en cero. De lo contrario, devuelve el valor true y establece el parámetro out en el valor numérico de la cadena.

> **NOTE**
>
> Una cadena puede contener solamente caracteres numéricos pero no ser válida para el tipo cuyo método `TryParse` se está usando. Por ejemplo, "256" no es un valor válido para `byte` pero sí para `int`. "98,6" no es un valor válido para `int` pero sí para `decimal`.

Ejemplo

En los ejemplos siguientes se muestra cómo usar `TryParse` con representaciones de cadena de los valores `long`, `byte` y `decimal`.

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
  Console.WriteLine("number1 now = {0}", number1);
else
  Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
  Console.WriteLine("number2 now = {0}", number2);
else
  Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
  Console.WriteLine("number3 now = {0}", number3);
else
  Console.WriteLine("number3 is not a valid decimal");
```

Programación sólida

Los tipos numéricos primitivos también implementan el método estático `Parse`, que produce una excepción si la cadena no es un número válido. `TryParse` es, en general, más eficaz porque simplemente devuelve false si el número no es válido.

Seguridad de .NET

Use siempre los métodos `TryParse` o `Parse` para validar los datos proporcionados por el usuario en controles como cuadros de texto y cuadros combinados.

Vea también

- Procedimiento Convertir una matriz de bytes en un valor int
- Procedimiento Convertir una cadena en un número
- Procedimiento Convertir cadenas hexadecimales en tipos numéricos
- Análisis de cadenas numéricas
- Aplicación de formato a tipos

Indizadores (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Los indizadores permiten indizar las instancias de una clase o struct como matrices. El valor indizado se puede establecer o recuperar sin especificar explícitamente un miembro de tipo o de instancia. Son similares a propiedades, excepto en que sus descriptores de acceso usan parámetros.

En el ejemplo siguiente se define una clase genérica con métodos de descriptor de acceso get y set sencillos para asignar y recuperar valores. La clase `Program` crea una instancia de esta clase para almacenar cadenas.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

NOTE

Para obtener más ejemplos, vea Secciones relacionadas.

Definiciones de cuerpos de expresión

Es habitual que un descriptor de acceso get o set de un indizador conste de una única instrucción que devuelve o establece un valor. Los miembros con forma de expresión proporcionan una sintaxis simplificada para admitir este escenario. A partir de C# 6, un indizador de solo lectura puede implementarse como un miembro con forma de expresión, como se muestra en el ejemplo siguiente.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Tenga en cuenta que `=>` presenta el cuerpo de la expresión y que la palabra clave `get` no se utiliza.

A partir de C# 7.0, los descriptores de acceso get y set se pueden implementar como miembros con forma de expresión. En este caso, sí deben utilizarse las palabras clave `get` y `set`. Por ejemplo:

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Información general sobre los indizadores

- Los indizadores permiten indizar objetos de manera similar a como se hace con las matrices.
- Un descriptor de acceso `get` devuelve un valor. Un descriptor de acceso `set` asigna un valor.
- La palabra clave this se usa para definir los indizadores.
- La palabra clave value se usa para definir el valor asignado por el indizador `set`.
- Los indizadores no tienen que ser indizados por un valor entero; depende de usted cómo definir el mecanismo de búsqueda concreto.
- Los indizadores se pueden sobrecargar.
- Los indizadores pueden tener más de un parámetro formal, por ejemplo, al tener acceso a una matriz bidimensional.

Secciones relacionadas

- Utilizar indizadores
- Indizadores en Interfaces
- Comparación entre propiedades e indizadores
- Restringir la accesibilidad del descriptor de acceso

Especificación del lenguaje C#

Para obtener más información, vea la sección Indizadores de Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Propiedades

Uso de indizadores (Guía de programación de C#)

18/03/2020 • 8 minutes to read • Edit Online

Los indexadores son una comodidad sintáctica que le permiten crear una clase, un struct o una interfaz a los que pueden acceder las aplicaciones cliente simplemente como una matriz. Los indexadores se implementan con más frecuencia en tipos cuyo propósito principal consiste en encapsular una matriz o colección interna. Por ejemplo, suponga que tiene una clase `TempRecord` que representa la temperatura en grados Fahrenheit que se registra en 10 momentos diferentes durante un período de 24 horas. La clase contiene una matriz `temps` de tipo `float[]` para almacenar los valores de temperatura. Si implementa un indizador en esta clase, los clientes pueden tener acceso a las temperaturas en una instancia de `TempRecord` como `float temp = tr[4]` en lugar de como `float temp = tr.temps[4]`. La notación del indexador no solo simplifica la sintaxis para las aplicaciones cliente; también hace que la clase y su finalidad sean más intuitivas para que las conozcan otros desarrolladores.

Para declarar un indizador en una clase o un struct, use la palabra clave this, como en este ejemplo:

```
public int this[int index]    // Indexer declaration
{
    // get and set accessors
}
```

Comentarios

Los tipos de un indexador y de sus parámetros deben ser al menos igual de accesibles que el propio indexador. Para obtener más información sobre los niveles de accesibilidad, vea Modificadores de acceso.

Para obtener más información sobre cómo usar los indexadores con una interfaz, vea Indizadores en interfaces.

La firma de un indexador consta del número y los tipos de sus parámetros formales. No incluye el tipo de indizador ni los nombres de los parámetros formales. Si declara más de un indexador en la misma clase, deben tener firmas diferentes.

Un valor de indexador no está clasificado como una variable; por tanto, no se puede pasar un valor de indexador como un parámetro ref u out.

Para proporcionar el indizador con un nombre que puedan usar otros lenguajes, use System.Runtime.CompilerServices.IndexerNameAttribute, como se muestra en el ejemplo siguiente:

```
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]   // Indexer declaration
{
    // get and set accessors
}
```

Este indexador tendrá el nombre `TheItem`. Si no se proporciona el atributo de nombre, `Item` será el nombre predeterminado.

Ejemplo 1

En el ejemplo siguiente, se muestra cómo declarar un campo de matriz privada, `temps`, como un indexador. El indexador permite el acceso directo a la instancia `tempRecord[i]`. La alternativa a usar el indexador es declarar la matriz como un miembro public y tener acceso directamente a sus miembros `tempRecord.temps[i]`.

Tenga en cuenta que, cuando se evalúa el acceso de un indexador (por ejemplo, en una instrucción `Console.Write`), se invoca al descriptor de acceso get. Por tanto, si no hay ningún descriptor de acceso `get`, se produce un error en tiempo de compilación.

```
class TempRecord
{
    // Array of temperature values
    private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
                                            61.3F, 65.9F, 62.1F, 59.2F, 57.5F };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length
    {
        get { return temps.Length; }
    }
    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get
        {
            return temps[index];
        }

        set
        {
            temps[index] = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        TempRecord tempRecord = new TempRecord();
        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
```

Indexación con otros valores

C# no limita el tipo de parámetro indizador a un entero. Por ejemplo, puede ser útil usar una cadena con un indexador. Este tipo de indexador podría implementarse al buscar la cadena de la colección y devolver el valor adecuado. Ya que los descriptores de acceso se pueden sobrecargar, las versiones de cadena y entero pueden coexistir.

Ejemplo 2

En el ejemplo siguiente se declara una clase que almacena los días de la semana. Un descriptor de acceso `get` toma una cadena, el nombre de un día, y devuelve el entero correspondiente. Por ejemplo, "Sunday" devuelve 0, "Monday" devuelve 1 y así sucesivamente.

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // Indexer with only a get accessor with the expression-bodied definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new System.ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported. Day input must be in the form \"Sun\", \"Mon\", etc");
    }
}

class Program
{
    static void Main()
    {
        var week = new DayCollection();
        System.Console.WriteLine(week["Fri"]);

        try
        {
            System.Console.WriteLine(week["Made-up day"]);
        }
        catch (System.ArgumentOutOfRangeException e)
        {
            System.Console.WriteLine($"Not supported input: {e.Message}");
        }
    }
    // Output:
    // 5
    // Not supported input: Day Made-up day is not supported. Day input must be in the form "Sun", "Mon", etc
(Parameter 'day')
}
```

Programación sólida

Hay dos formas principales en que se pueden mejorar la seguridad y confiabilidad de los indexadores:

- Asegúrese de incorporar algún tipo de estrategia de control de errores para controlar la posibilidad de que el código de cliente pase un valor de índice no válido. En el primer ejemplo de este tema, la clase TempRecord proporciona una propiedad Length que permite al código de cliente comprobar la entrada antes de pasarla al indexador. También puede colocar el código de control de errores en el propio indexador. Asegúrese de documentar para los usuarios cualquier excepción que se produzca dentro de un descriptor de acceso del indexador.

- Establezca la accesibilidad de los descriptores de acceso get and set para que sea tan restrictiva como razonable. Esto es importante para el descriptor de acceso `set` en particular. Para más información, vea Restringir la accesibilidad del descriptor de acceso.

Vea también

- Guía de programación de C#
- Indizadores
- Propiedades

Indizadores en interfaces (Guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

Los indexadores se pueden declarar en una interfaz. Los descriptores de acceso de los indexadores de interfaz se diferencian de los descriptores de acceso de los indexadores de clase de las maneras siguientes:

- Los descriptores de acceso de interfaz no usan modificadores.
- Normalmente, un descriptor de acceso de interfaz no tiene un cuerpo.

El propósito del descriptor de acceso es indicar si el indizador es de lectura y escritura, de solo lectura o de solo escritura. Puede proporcionar una implementación para un indizador definido en una interfaz, pero esto es poco frecuente. Los indizadores suelen definir una API para acceder a los campos de datos y los campos de datos no se pueden definir en una interfaz.

A continuación tiene un ejemplo de un descriptor de acceso de indexador de interfaz:

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

La firma de un indexador debe ser diferente de las firmas de los demás indexadores declarados en la misma interfaz.

Ejemplo

En el siguiente ejemplo, se muestra cómo implementar indexadores de interfaz.

```
// Indexer on an interface:
public interface IIndexInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        // The arr object will throw IndexOutOfRange exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

```
IndexerClass test = new IndexerClass();
System.Random rand = new System.Random();
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/
```

En el ejemplo anterior, podría usar la implementación del miembro de interfaz explícita al usar el nombre completo del miembro de interfaz. Por ejemplo

```
string IIndexInterface.this[int index]
{
}
```

En cambio, el nombre completo solo es necesario para evitar la ambigüedad cuando la clase implementa más de una interfaz con la misma firma de indexador. Por ejemplo, si una clase `Employee` implementa dos interfaces `ICitizen` y `IEmployee` y ambas interfaces tienen la misma firma de indexador, la implementación del miembro de interfaz explícita es necesaria. Es decir, la siguiente declaración de indexador:

```
string IEmployee.this[int index]
{
}
```

implementa el indexador en la interfaz `IEmployee`, mientras que la siguiente declaración:

```
string ICitizen.this[int index]
{
}
```

implementa el indexador en la interfaz `ICitizen`.

Vea también

- Guía de programación de C#
- Indizadores
- Propiedades
- Interfaces

Comparación entre propiedades e indizadores (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Los indexadores son como propiedades. Excepto por las diferencias que se muestran en la tabla siguiente, todas las reglas que se definen para los descriptores de acceso de propiedad se aplican también a los descriptores de acceso de indexador.

PROPERTY	INDEXADOR
Permite que los métodos se llamen como si fueran miembros de datos públicos.	Permite que se pueda tener acceso a los elementos de una colección interna de un objeto mediante la notación de matriz en el propio objeto.
Se ha tenido acceso mediante un nombre simple.	Se ha tenido acceso mediante un índice.
Puede ser un miembro de instancia o estático.	Debe ser un miembro de instancia.
Un descriptor de acceso get de una propiedad no tiene parámetros.	Un descriptor de acceso `get` de un indexador tiene la misma lista de parámetros formales que el indexador.
Un descriptor de acceso set de una propiedad contiene el parámetro `value` implícito.	Un descriptor de acceso `set` de un indexador tiene la misma lista de parámetros formales que el indexador, y también para el parámetro value.
Admite la sintaxis abreviada con Propiedades autoimplementadas.	Admite miembros de cuerpo de expresión para obtener solo indexadores.

Vea también

- Guía de programación de C#
- Indizadores
- Propiedades

Eventos (Guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

Cuando ocurre algo interesante, los eventos habilitan una clase u objeto para notificarlo a otras clases u objetos. La clase que envía (o *genera*) el evento recibe el nombre de *publicador* y las clases que reciben (o *controlan*) el evento se denominan *suscriptores*.

En una aplicación web o una aplicación de Windows Forms en C# típica, se puede suscribir a eventos generados por controles, como botones y cuadros de lista. Puede usar el entorno de desarrollo integrado (IDE) de Visual C# para examinar los eventos que publica un control y seleccionar los que quiera administrar. El IDE proporciona una forma sencilla de agregar automáticamente un método de controlador de eventos vacío y el código para suscribirse al evento. Para obtener más información, vea Procedimiento para suscribir y cancelar la suscripción a eventos.

Información general sobre eventos

Los eventos tienen las siguientes propiedades:

- El publicador determina el momento en el que se genera un evento; los suscriptores determinan la acción que se lleva a cabo en respuesta al evento.
- Un evento puede tener varios suscriptores. Un suscriptor puede controlar varios eventos de varios publicadores.
- Nunca se generan eventos que no tienen suscriptores.
- Los eventos se suelen usar para indicar acciones del usuario, como los clics de los botones o las selecciones de menú en las interfaces gráficas de usuario.
- Cuando un evento tiene varios suscriptores, los controladores de eventos se invocan sincrónicamente cuando se genera un evento. Para invocar eventos de forma asincrónica, consulte Llamar a métodos sincrónicos de forma asincrónica.
- En la biblioteca de clases de .NET Framework, los eventos se basan en el delegado EventHandler y en la clase base EventArgs.

Secciones relacionadas

Para obtener más información, consulte:

- Procedimiento para suscribir y cancelar la suscripción a eventos
- Procedimiento para publicar eventos que cumplan las instrucciones de .NET
- Procedimiento para generar eventos de una clase base en clases derivadas
- Procedimiento para implementar eventos de interfaz
- Procedimiento para implementar descriptores de acceso de eventos personalizados

Especificación del lenguaje C#

Para obtener más información, consulte la sección Eventos de Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Capítulos destacados del libro

Delegates, Events, and Lambda Expressions (Delegados, eventos y expresiones lambda) en C# 3.0 Cookbook, Tercera edición: More than 250 solutions for C# 3.0 programmers (Más de 250 soluciones para programadores de C# 3.0)

Delegates and Events (Delegados y eventos) en Learning C# 3.0: Master the fundamentals of C# 3.0

Vea también

- EventHandler
- Guía de programación de C#
- Delegados
- Crear controladores de eventos en Windows Forms

Procedimiento Suscribir y cancelar la suscripción a eventos (Guía de programación de C#)

29/06/2020 • 7 minutes to read • Edit Online

La suscripción a un evento publicado por otra clase se realiza cuando quiere escribir código personalizado al que se llama cuando se produce ese evento. Por ejemplo, puede suscribirse al evento `click` de un botón para que la aplicación realice alguna operación cuando el usuario haga clic en el botón.

Para suscribirse a eventos mediante el IDE de Visual Studio

1. Si no puede ver la ventana **Propiedades**, en la vista **Diseño** haga clic con el botón derecho en el formulario o control para el que quiere crear un controlador de eventos y seleccione **Propiedades**.

2. En la parte superior de la ventana **Propiedades**, haga clic en el icono **Eventos**.

3. Haga doble clic en el evento que quiera crear, por ejemplo, el evento `Load`.

   Visual C# crea un método de controlador de eventos vacío y lo agrega al código. También puede agregar manualmente el código en la vista **Código**. Por ejemplo, las líneas siguientes de código declaran un método de controlador de eventos al que se llamará cuando la clase `Form` genere el evento `Load`.

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   La línea de código que es necesaria para suscribirse al evento también se genera automáticamente con el método `InitializeComponent` en el archivo Form1.Designer.cs del proyecto. Se parece a lo siguiente:

   ```
   this.Load += new System.EventHandler(this.Form1_Load);
   ```

Para suscribirse a eventos mediante programación

1. Defina un método de controlador de eventos cuya firma coincida con la firma de delegado del evento. Por ejemplo, si el evento se basa en el tipo de delegado EventHandler, el siguiente código representa el código auxiliar del método:

   ```
   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Use el operador de suma y asignación (`+=`) para asociar el controlador de eventos al evento. En el ejemplo siguiente, se asume que un objeto denominado `publisher` tiene un evento denominado `RaiseCustomEvent`. Observe que la clase de suscriptor necesita una referencia a la clase de editor para suscribirse a sus eventos.

   ```
   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

   Observe que la sintaxis anterior es nueva en C# 2.0. Al igual que en la sintaxis de C# 1.0, el delegado

encapsulador debe crearse explícitamente mediante la palabra clave `new`:

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCustomEvent);
```

También puede usar una expresión lambda para especificar un controlador de eventos:

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
        {
            MessageBox.Show(((MouseEventArgs)e).Location.ToString());
        };
}
```

Para suscribirse a eventos mediante un método anónimo

- Si no tiene que cancelar la suscripción a un evento más adelante, puede usar el operador de suma y asignación (`+=`) para asociar un método anónimo al evento. En el ejemplo siguiente, se presupone que un objeto denominado `publisher` tiene un evento denominado `RaiseCustomEvent` y que se ha definido una clase `CustomEventArgs` para proporcionar algún tipo de información específica del evento. Observe que la clase de suscriptor necesita una referencia a `publisher` para suscribirse a sus eventos.

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArgs e)
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

Es importante tener en cuenta que puede no resultar fácil cancelar la suscripción a un evento si se ha usado una función anónima para suscribirse a él. Para cancelar la suscripción en esta situación, es necesario regresar al código donde se ha suscrito al evento, almacenar el método anónimo en una variable de delegado y, después, agregar el delegado al evento. En general, se recomienda que no use funciones anónimas para suscribirse a eventos si va a tener que cancelar la suscripción al evento en el código más adelante. Para obtener más información sobre las funciones anónimas, vea Funciones anónimas.

Cancelar una suscripción

Para impedir que se invoque el controlador de eventos cuando se produce el evento, puede cancelar la suscripción al evento. Para evitar que se pierdan recursos, debe cancelar la suscripción a los eventos antes de eliminar un objeto suscriptor. Hasta que se cancela la suscripción a un evento, el delegado multidifusión subyacente al evento en el objeto de publicación tiene una referencia al delegado que encapsula el controlador de eventos del suscriptor. Mientras el objeto de publicación mantenga esa referencia, la recolección de elementos no utilizados no eliminará el objeto suscriptor.

Para cancelar la suscripción a un evento

- Use el operador de resta y asignación (`-=`) para cancelar la suscripción a un evento:

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

Cuando se haya cancelado la suscripción a un evento de todos los suscriptores, la instancia del evento en la clase de editor se establecerá en `null`.

Vea también

- Eventos
- event
- Procedimiento para publicar eventos que cumplan las instrucciones de .NET
- Operadores - y -=
- Operadores + y +=

Procedimiento Publicar eventos que cumplan las directrices de .NET (guía de programación de C#)

29/06/2020 • 5 minutes to read • Edit Online

En el siguiente procedimiento se muestra cómo agregar eventos que cumplan el patrón de .NET estándar a las clases y structs. Todos los eventos de la biblioteca de clases de .NET Framework se basan en el delegado EventHandler, que se define de la siguiente manera:

```
public delegate void EventHandler(object sender, EventArgs e);
```

> **NOTE**
> .NET Framework 2.0 incluye una versión genérica de este delegado, EventHandler<TEventArgs>. En los siguientes ejemplos se muestra cómo usar las dos versiones.

Aunque los eventos de las clases que defina se pueden basar en cualquier tipo de delegado válido, incluidos los delegados que devuelven un valor, por lo general se recomienda que base los eventos en el patrón de .NET mediante EventHandler, como se muestra en el ejemplo siguiente.

El nombre `EventHandler` puede dar lugar a un poco de confusión, ya que realmente no controla el evento. EventHandler y EventHandler<TEventArgs> genérico son tipos de delegado. Un método o una expresión lambda cuya firma coincide con la definición de delegado es el *controlador de eventos* y se invocará cuando se genere el evento.

Publicación de eventos basados en el patrón EventHandler

1. (Omita este paso y vaya al paso 3a si no tiene que enviar datos personalizados con el evento). Declare la clase de los datos personalizados en un ámbito que sea visible para las clases de publicador y suscriptor. Luego, agregue los miembros necesarios para almacenar los datos de eventos personalizados. En este ejemplo se devuelve una cadena simple.

   ```
   public class CustomEventArgs : EventArgs
   {
       public CustomEventArgs(string message)
       {
           Message = message;
       }

       public string Message { get; set; }
   }
   ```

2. (Omita este paso si usa la versión genérica de EventHandler<TEventArgs>). Declare un delegado en la clase de publicación. Asígnele un nombre que termine con `EventHandler`. El segundo parámetro especifica el tipo `EventArgs` personalizado.

   ```
   public delegate void CustomEventHandler(object sender, CustomEventArgs args);
   ```

3. Declare el evento en la clase de publicación llevando a cabo uno de los siguientes pasos.

a. Si no tiene ninguna clase EventArgs personalizada, el tipo Event será el delegado EventHandler no genérico. No es necesario declarar el delegado, porque ya está declarado en el espacio de nombres System que se incluye al crear el proyecto de C#. Agregue el código siguiente a la clase de publicador.

```
public event EventHandler RaiseCustomEvent;
```

b. Si usa la versión no genérica de EventHandler y tiene una clase personalizada derivada de EventArgs, declare el evento dentro de la clase de publicación y use el delegado del paso 2 como tipo.

```
public event CustomEventHandler RaiseCustomEvent;
```

c. Si usa la versión genérica, no necesita ningún delegado personalizado. En su lugar, en la clase de publicación, especifique el tipo de evento como `EventHandler<CustomEventArgs>`, sustituyendo el nombre de su propia clase que aparece entre corchetes angulares.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

Ejemplo

En el siguiente ejemplo se muestran los pasos anteriores mediante el uso de una clase EventArgs personalizada y EventHandler<TEventArgs> como tipo de evento.

```
using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
```

```
            // immediately after the null check and before the event is raised.
            EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (raiseEvent != null)
            {
                // Format the string to send inside the CustomEventArgs parameter
                e.Message += $" at {DateTime.Now}";

                // Call to raise the event.
                raiseEvent(this, e);
            }
        }
    }

    //Class that subscribes to an event
    class Subscriber
    {
        private readonly string _id;

        public Subscriber(string id, Publisher pub)
        {
            _id = id;

            // Subscribe to the event
            pub.RaiseCustomEvent += HandleCustomEvent;
        }

        // Define what actions to take when the event is raised.
        void HandleCustomEvent(object sender, CustomEventArgs e)
        {
            Console.WriteLine($"{_id} received this message: {e.Message}");
        }
    }

    class Program
    {
        static void Main()
        {
            var pub = new Publisher();
            var sub1 = new Subscriber("sub1", pub);
            var sub2 = new Subscriber("sub2", pub);

            // Call the method that raises the event.
            pub.DoSomething();

            // Keep the console window open
            Console.WriteLine("Press any key to continue...");
            Console.ReadLine();
        }
    }
}
```

Vea también

- Delegate
- Guía de programación de C#
- Eventos
- Delegados

Procedimiento Producir eventos de una clase base en clases derivadas (Guía de programación de C#)

29/06/2020 • 4 minutes to read • Edit Online

En el siguiente ejemplo sencillo se muestra la forma estándar de declarar eventos en una clase base para que también se puedan generar desde clases derivadas. Este patrón se usa mucho en las clases de Windows Forms de las bibliotecas de clases de .NET.

Al crear una clase que se pueda usar como clase base para otras clases, debe considerar el hecho de que los eventos son un tipo especial de delegado que solo se pueden invocar desde la clase que los haya declarado. Las clases derivadas no pueden invocar directamente a eventos declarados en la clase base. Aunque a veces pueda querer un evento que solo la clase base pueda generar, en la mayoría de los casos debería habilitar la clase derivada para invocar a eventos de clase base. Para ello, puede crear un método de invocación protegido en la clase base que encapsula el evento. Al llamar o invalidar a este método de invocación, las clases derivadas pueden invocar directamente al evento.

> **NOTE**
> No declare eventos virtuales en una clase base y los invalide en una clase derivada. El compilador de C# no los controla correctamente y no es posible decir si un suscriptor del evento derivado se está suscribiendo realmente al evento de clase base.

Ejemplo

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();
```

```
        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }

    public class Rectangle : Shape
    {
        private double _length;
        private double _width;

        public Rectangle(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
        }

        public void Update(double length, double width)
        {
            _length = length;
            _width = width;
            _area = _length * _width;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any rectangle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
```

```
            Console.WriteLine("Drawing a rectangle");
        }
    }

    // Represents the surface on which the shapes are drawn
    // Subscribes to shape events so that it knows
    // when to redraw a shape.
    public class ShapeContainer
    {
        private readonly List<Shape> _list;

        public ShapeContainer()
        {
            _list = new List<Shape>();
        }

        public void AddShape(Shape shape)
        {
            _list.Add(shape);

            // Subscribe to the base class event.
            shape.ShapeChanged += HandleShapeChanged;
        }

        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now {e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
 */
```

Vea también

- Guía de programación de C#
- Eventos
- Delegados
- Modificadores de acceso
- Crear controladores de eventos en Windows Forms

Procedimiento Implementar eventos de interfaz (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Un interfaz puede declarar un evento. En el siguiente ejemplo, se muestra cómo implementar eventos de interfaz en una clase. Básicamente, las reglas son las mismas que para implementar cualquier propiedad o método de interfaz.

Para implementar eventos de interfaz en una clase

Declare el evento en la clase y, después, invóquelo en las áreas apropiadas.

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Ejemplo

En el ejemplo siguiente se muestra cómo controlar la situación menos común en que la clase hereda de dos o más interfaces y cada interfaz tiene un evento con el mismo nombre. En esta situación, debe proporcionar una implementación de interfaz explícita para uno de los eventos como mínimo. Al escribir una implementación de interfaz explícita para un evento, también debe escribir los descriptores de acceso del evento `add` y `remove`. Normalmente, los proporciona el compilador, pero en este caso no es posible.

Al proporcionar sus propios descriptores de acceso, puede especificar si los dos eventos se representan mediante el mismo evento en la clase o mediante eventos diferentes. Por ejemplo, si los eventos deben provocarse en momentos diferentes según las especificaciones de la interfaz, puede asociar cada evento a una implementación distinta en su clase. En el ejemplo siguiente, los suscriptores determinan qué evento `OnDraw` recibirán al convertir la referencia de forma en `IShape` o en `IDrawingObject`.

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
```

```
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

Vea también

- Guía de programación de C#
- Eventos
- Delegados
- Implementación de interfaz explícita
- Procedimiento para generar eventos de una clase base en clases derivadas

Procedimiento Implementar descriptores de acceso de eventos personalizados (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Un evento es un tipo especial de delegado de multidifusión que solo puede invocarse desde dentro de la clase en la que se declara. El código cliente se suscribe al evento proporcionando una referencia a un método que debería invocarse cuando se desencadena el evento. Estos métodos se agregan a la lista de invocación del delegado a través de descriptores de acceso de eventos, que son similares a los descriptores de acceso de propiedad, con la diferencia de que los descriptores de acceso de eventos se denominan `add` y `remove`. En la mayoría de los casos, no tiene que proporcionar descriptores de acceso de eventos personalizados. Cuando no proporciona ningún descriptor de acceso de eventos personalizado en el código, el compilador los agrega automáticamente. En cambio, en algunos casos puede que tenga que proporcionar un comportamiento personalizado. Uno de estos casos se muestra en el tema Procedimiento Implementar eventos de interfaz.

Ejemplo

En el ejemplo siguiente se muestra cómo implementar descriptores de acceso de eventos add y remove personalizados. Aunque puede sustituir cualquier código dentro de los descriptores de acceso, recomendamos que bloquee el evento antes de agregar o quitar un nuevo método de control de eventos.

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

Vea también

- Eventos
- event

Genéricos (Guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

Los genéricos introducen el concepto de parámetros de tipo a .NET, lo que le permite diseñar clases y métodos que aplazan la especificación de uno o varios tipos hasta que el código de cliente declare y cree una instancia de la clase o el método. Por ejemplo, al usar un parámetro de tipo genérico `T`, puede escribir una clase única que otro código de cliente puede usar sin incurrir en el costo o riesgo de conversiones en tiempo de ejecución u operaciones de conversión boxing, como se muestra aquí:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();
        list2.Add("");

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
        list3.Add(new ExampleClass());
    }
}
```

Las clases y métodos genéricos combinan reusabilidad, seguridad de tipos y eficacia de una manera en que sus homólogos no genéricos no pueden. Los genéricos se usan frecuentemente con colecciones y los métodos que funcionan en ellas. El espacio de nombres System.Collections.Generic contiene varias clases de colecciones basadas en genéricos. No se recomiendan las colecciones no genéricas, como ArrayList, y se mantienen por compatibilidad. Para más información, vea Elementos genéricos en .NET.

Por supuesto, también se pueden crear tipos y métodos genéricos personalizados para proporcionar soluciones y patrones de diseño generalizados propios con seguridad de tipos y eficaces. En el ejemplo de código siguiente se muestra una clase genérica simple de lista vinculada para fines de demostración. (En la mayoría de los casos, debe usar la clase List<T> proporcionada por .NET en lugar de crear la suya propia). El parámetro de tipo `T` se usa en diversas ubicaciones donde normalmente se usaría un tipo concreto para indicar el tipo del elemento almacenado en la lista. Se usa de estas formas:

- Como el tipo de un parámetro de método en el método `AddHead`.
- Como el tipo de valor devuelto de la propiedad `Data` en la clase anidada `Node`.
- Como el tipo de miembro privado `data` de la clase anidada.

Tenga en cuenta que `T` está disponible para la clase anidada `Node`. Cuando se crea una instancia de `GenericList<T>` con un tipo concreto, por ejemplo como un `GenericList<int>`, cada repetición de `T` se sustituye por `int`.

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node next;
        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
    public void AddHead(T t)
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

En el ejemplo de código siguiente se muestra cómo el código de cliente usa la clase genérica `GenericList<T>` para crear una lista de enteros. Simplemente cambiando el argumento de tipo, el código siguiente puede modificarse fácilmente para crear listas de cadenas o cualquier otro tipo personalizado:

```
class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}
```

Introducción a los genéricos

- Use tipos genéricos para maximizar la reutilización del código, la seguridad de tipos y el rendimiento.
- El uso más común de los genéricos es crear clases de colección.
- La biblioteca de clases de .NET contiene varias clases de colección genéricas en el espacio de nombres System.Collections.Generic. Estas se deberían usar siempre que sea posible en lugar de clases como ArrayList en el espacio de nombres System.Collections.
- Puede crear sus propias interfaces, clases, métodos, eventos y delegados genéricos.
- Puede limitar las clases genéricas para habilitar el acceso a métodos en tipos de datos determinados.
- Puede obtener información sobre los tipos que se usan en un tipo de datos genérico en tiempo de ejecución mediante la reflexión.

Secciones relacionadas

- Parámetros de tipo genérico
- Restricciones de tipos de parámetros
- Clases genéricas
- Interfaces genéricas
- Métodos genéricos
- Delegados genéricos
- Diferencias entre plantillas de C++ y tipos genéricos de C#
- Genéricos y reflexión
- Genéricos en el motor en tiempo de ejecución

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#.

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Tipos
- <typeparam>

- <typeparamref>
- Elementos genéricos en .NET

Parámetros de tipos genéricos (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

En un tipo genérico o en una definición de método, un parámetro de tipo es un marcador de posición para un tipo específico que un cliente especifica cuando crean instancias de una variable del tipo genérico. Una clase genérica, como `GenericList<T>` que se muestra en Introducción a los genéricos, no puede usarse como está porque no es realmente un tipo; es más parecida a un plano para un tipo. Para usar `GenericList<T>`, el código cliente debe declarar y crear instancias de un tipo construido especificando un argumento de tipo dentro de corchetes angulares. El argumento de tipo de esta clase determinada puede ser cualquier tipo reconocido por el compilador. Puede crearse cualquier número de instancias de tipo construidas, usando cada una un argumento de tipo diferente, de la manera siguiente:

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

En cada una de estas instancias de `GenericList<T>`, cada aparición de `T` en la clase se sustituye en tiempo de ejecución con el argumento de tipo. Mediante esta sustitución, hemos creado tres objetos eficaces y con seguridad de tipos independientes con una definición de clase única. Para obtener más información sobre cómo se realiza esta sustitución mediante CLR, vea Genéricos en tiempo de ejecución.

Instrucciones de nomenclatura de los parámetros de tipo

- **Asigne** nombres descriptivos a los parámetros de tipo genérico, a no ser que un nombre de una sola letra sea completamente claro y un nombre descriptivo no agregue ningún valor.

  ```
  public interface ISessionChannel<TSession> { /*...*/ }
  public delegate TOutput Converter<TInput, TOutput>(TInput from);
  public class List<T> { /*...*/ }
  ```

- **Considere** el uso de T como el nombre del parámetro de tipo para los tipos con un parámetro de tipo de una sola letra.

  ```
  public int IComparer<T>() { return 0; }
  public delegate bool Predicate<T>(T item);
  public struct Nullable<T> where T : struct { /*...*/ }
  ```

- **Establezca** el prefijo "T" a los nombres de parámetro de tipo descriptivos.

  ```
  public interface ISessionChannel<TSession>
  {
      TSession Session { get; }
  }
  ```

- **Considere** la posibilidad de indicar restricciones que se encuentran en un parámetro de tipo en el nombre del parámetro. Por ejemplo, un parámetro restringido a `ISession` puede llamarse `TSession`.

La regla de análisis de código CA1715 puede usarse para asegurarse de que los parámetros de tipo se denominan

apropiadamente.

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Genéricos
- Diferencias entre plantillas de C++ y tipos genéricos de C#

Restricciones de tipos de parámetros (Guía de programación de C#)

29/06/2020 • 19 minutes to read • Edit Online

Las restricciones informan al compilador sobre las capacidades que debe tener un argumento de tipo. Sin restricciones, el argumento de tipo puede ser cualquier tipo. El compilador solo puede suponer los miembros de System.Object, que es la clase base fundamental de los tipos .NET. Para más información, vea Por qué usar restricciones. Si el código de cliente usa un tipo que no cumple una restricción, el compilador emite un error. Las restricciones se especifican con la palabra clave contextual `where`. En la tabla siguiente se muestran los siete tipos de restricciones:

RESTRICCIÓN	DESCRIPCIÓN
`where T : struct`	El argumento de tipo debe ser un tipo que no acepta valores NULL. Para más información sobre los tipos de valor que admiten un valor NULL, consulte Tipos de valor que admiten un valor NULL. Todos los tipos de valor tienen un constructor sin parámetros accesible, por lo que la restricción `struct` implica la restricción `new()` y no se puede combinar con la restricción `new()`. No puede combinar la restricción `struct` con la restricción `unmanaged`.
`where T : class`	El argumento de tipo debe ser un tipo de referencia. Esta restricción se aplica también a cualquier clase, interfaz, delegado o tipo de matriz. En un contexto que admite un valor NULL en C# 8.0 o versiones posteriores, `T` debe ser un tipo de referencia que no acepte valores NULL.
`where T : class?`	El argumento de tipo debe ser un tipo de referencia, que acepte o no valores NULL. Esta restricción se aplica también a cualquier clase, interfaz, delegado o tipo de matriz.
`where T : notnull`	El argumento de tipo debe ser un tipo que no acepta valores NULL. El argumento puede ser un tipo de referencia que no acepta valores NULL en C# 8.0 o posterior, o bien un tipo de valor que no acepta valores NULL.
`where T : unmanaged`	El argumento de tipo debe ser un tipo no administrado que no acepta valores NULL. La restricción `unmanaged` implica la restricción `struct` y no se puede combinar con las restricciones `struct` ni `new()`.
`where T : new()`	El argumento de tipo debe tener un constructor sin parámetros público. Cuando se usa conjuntamente con otras restricciones, la restricción `new()` debe especificarse en último lugar. La restricción `new()` no se puede combinar con las restricciones `struct` ni `unmanaged`.

RESTRICCIÓN	DESCRIPCIÓN
`where T :` *<base class name>*	El argumento de tipo debe ser o derivarse de la clase base especificada. En un contexto que admite un valor NULL en C# 8.0 y versiones posteriores, `T` debe ser un tipo de referencia que no acepta valores NULL derivado de la clase base especificada.
`where T :` *<base class name>?*	El argumento de tipo debe ser o derivarse de la clase base especificada. En un contexto que admite un valor NULL en C# 8.0 y versiones posteriores, `T` puede ser un tipo que acepta o no acepta valores NULL derivado de la clase base especificada.
`where T :` *<interface name>*	El argumento de tipo debe ser o implementar la interfaz especificada. Pueden especificarse varias restricciones de interfaz. La interfaz de restricciones también puede ser genérica. En un contexto que admite un valor NULL en C# 8.0 y versiones posteriores, `T` debe ser un tipo que no acepta valores NULL que implementa la interfaz especificada.
`where T :` *<interface name>?*	El argumento de tipo debe ser o implementar la interfaz especificada. Pueden especificarse varias restricciones de interfaz. La interfaz de restricciones también puede ser genérica. En un contexto que admite un valor NULL en C# 8.0, `T` puede ser un tipo de referencia que admite un valor NULL, un tipo de referencia que no acepta valores NULL o un tipo de valor. `T` no puede ser un tipo de valor que admite un valor NULL.
`where T : U`	El argumento de tipo proporcionado por `T` debe ser o se debe derivar del argumento proporcionado para `U`. En un contexto que admite un valor NULL, si `U` puede ser un tipo de referencia que no acepta valores NULL, `T` debe ser un tipo de referencia que no acepta valores NULL. Si `U` es un tipo de referencia que admite un valor NULL, `T` puede aceptar valores NULL o no.

Por qué usar restricciones

Las restricciones especifican las funciones y expectativas de un parámetro de tipo. La declaración de esas restricciones significa que puede usar las operaciones y las llamadas de método del tipo de restricción. Si la clase o el método genérico usa cualquier operación en los miembros genéricos más allá de una asignación simple o una llamada a un método que System.Object no admita, tendrá que aplicar restricciones al parámetro de tipo. Por ejemplo, la restricción de clase base indica al compilador que solo los objetos de este tipo o derivados de este tipo se usarán como argumentos de tipo. Una vez que el compilador tenga esta garantía, puede permitir que los métodos de ese tipo se llamen en la clase genérica. En el ejemplo de código siguiente se muestran las funciones que podemos agregar a la clase `GenericList<T>` (en Introducción a los genéricos) mediante la aplicación de una restricción de clase base.

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node Next { get; set; }
        public T Data { get; set; }
    }

    private Node head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T FindFirstOccurrence(string s)
    {
        Node current = head;
        T t = null;

        while (current != null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                t = current.Data;
                break;
            }
            else
            {
                current = current.Next;
            }
        }
        return t;
    }
}
```

La restricción permite que la clase genérica use la propiedad `Employee.Name`. La restricción especifica que está garantizado que todos los elementos de tipo `T` sean un objeto `Employee` u objeto que hereda de `Employee`.

Pueden aplicarse varias restricciones en el mismo parámetro de tipo, y las propias restricciones pueden ser tipos genéricos, de la manera siguiente:

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

Al aplicar la restricción `where T : class`, evite los operadores `==` y `!=` en el parámetro de tipo porque estos operadores se probarán solo para la identidad de referencia, no para la igualdad de valor. Este comportamiento se produce incluso si estos operadores están sobrecargados en un tipo que se usa como un argumento. En el código siguiente se ilustra este punto; el resultado es False incluso cuando la clase String sobrecarga al operador `==`.

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
    string s2 = sb.ToString();
    OpEqualsTest<string>(s1, s2);
}
```

El compilador solo sabe que `T` es un tipo de referencia en tiempo de compilación y debe usar los operadores predeterminados que son válidos para todos los tipos de referencia. Si debe probar la igualdad de valor, la manera recomendada también es aplicar la restricción `where T : IEquatable<T>` o `where T : IComparable<T>` e implementar esa interfaz en cualquier clase que se usará para construir la clase genérica.

Restringir varios parámetros

Puede aplicar restricciones a varios parámetros, y varias restricciones a un solo parámetro, como se muestra en el siguiente ejemplo:

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Parámetros de tipo sin enlazar

Los parámetros de tipo que no tienen restricciones, como T en la clase pública `SampleClass<T>{}`, se denominan parámetros de tipo sin enlazar. Los parámetros de tipo sin enlazar tienen las reglas siguientes:

- Los operadores `!=` y `==` no pueden usarse porque no existe ninguna garantía de que el argumento de tipo concreto admita estos operadores.
- Pueden convertirse a y desde `System.Object` o convertirse explícitamente en cualquier tipo de interfaz.
- Puede compararlos con NULL. Si un parámetro sin enlazar se compara con `null`, la comparación siempre devolverá False si el argumento de tipo es un tipo de valor.

Parámetros de tipo como restricciones

El uso de un parámetro de tipo genérico como una restricción es útil cuando una función de miembro con su propio parámetro de tipo tiene que restringir ese parámetro al parámetro de tipo del tipo contenedor, como se

muestra en el ejemplo siguiente:

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

En el ejemplo anterior, `T` es una restricción de tipo en el contexto del método `Add`, y un parámetro de tipo sin enlazar en el contexto de la clase `List`.

Los parámetros de tipo también pueden usarse como restricciones en definiciones de clase genéricas. El parámetro de tipo debe declararse dentro de los corchetes angulares junto con los demás parámetros de tipo:

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

La utilidad de los parámetros de tipo como restricciones con clases genéricas es limitada, ya que el compilador no puede dar por supuesto nada sobre el parámetro de tipo, excepto que deriva de `System.Object`. Use parámetros de tipo como restricciones en clases genéricas en escenarios en los que quiere aplicar una relación de herencia entre dos parámetros de tipo.

Restricción NotNull

A partir de C# 8.0 en un contexto que admite un valor NULL, puede usar la restricción `notnull` para especificar que el argumento de tipo debe ser un tipo de valor que no acepta valores NULL o un tipo de referencia que no acepta valores NULL. La restricción `notnull` solo se puede usar en un contexto `nullable enable`. El compilador genera una advertencia si se agrega una restricción `notnull` en un contexto donde se desconoce que se aceptan valores NULL.

A diferencia de otras restricciones, cuando un argumento de tipo infringe la restricción `notnull`, el compilador genera una advertencia cuando el código se compila en un contexto `nullable enable`. Si el código se compila en un contexto donde se desconoce que se aceptan valores NULL, el compilador no genera ninguna advertencia o error.

A partir de C# 8.0 en un contexto que admite un valor NULL, la restricción `class` especifica que el argumento de tipo debe ser un tipo de referencia que no acepta valores NULL. En un contexto que admite un valor NULL, cuando un parámetro de tipo es un tipo de referencia que admite un valor NULL, el compilador genera una advertencia.

Restricción no administrada

A partir de C# 7.3, puede usar la restricción `unmanaged` para especificar que el parámetro de tipo debe ser un tipo no administrado que no acepta valores NULL. La restricción `unmanaged` permite escribir rutinas reutilizables para trabajar con tipos que se pueden manipular como bloques de memoria, como se muestra en el ejemplo siguiente:

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

El método anterior debe compilarse en un contexto `unsafe`, ya que usa el operador `sizeof` en un tipo que se desconoce si es integrado. Sin la restricción `unmanaged`, el operador `sizeof` no está disponible.

La restricción `unmanaged` implica la restricción `struct` y no se puede combinar con ella. Dado que la restricción `struct` implica la restricción `new()`, la restricción `unmanaged` tampoco se puede combinar con la restricción `new()`.

Restricciones de delegado

También a partir de C# 7.3, puede usar System.Delegate o System.MulticastDelegate como una restricción de clase base. CLR siempre permitía esta restricción, pero el lenguaje C# no la permitía. La restricción `System.Delegate` permite escribir código que funciona con los delegados en un modo con seguridad de tipos. En el código siguiente se define un método de extensión que combina dos delegados siempre y cuando sean del mismo tipo:

```
public static TDelegate TypeSafeCombine<TDelegate>(this TDelegate source, TDelegate target)
    where TDelegate : System.Delegate
    => Delegate.Combine(source, target) as TDelegate;
```

Puede usar el método anterior para combinar delegados que sean del mismo tipo:

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

Si quita la marca de comentario de la última línea, no se compilará. Tanto `first` como `test` son tipos de delegado, pero son tipos de delegado distintos.

Restricciones de enumeración

A partir de C# 7.3, también puede especificar el tipo System.Enum como una restricción de clase base. CLR siempre permitía esta restricción, pero el lenguaje C# no la permitía. Los genéricos que usan `System.Enum` proporcionan programación con seguridad de tipos para almacenar en caché los resultados de usar los métodos estáticos en `System.Enum`. En el ejemplo siguiente se buscan todos los valores válidos para un tipo de enumeración y, después, se compila un diccionario que asigna esos valores a su representación de cadena.

```
public static Dictionary<int, string> EnumNamedValues<T>() where T : System.Enum
{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item));
    return result;
}
```

`Enum.GetValues` y `Enum.GetName` usan reflexión, lo que tiene consecuencias en el rendimiento. Puede llamar a `EnumNamedValues` para compilar una recopilación que se almacene en caché y se vuelva a usar, en lugar de repetir las llamadas que requieren reflexión.

Podría usarla como se muestra en el ejemplo siguiente para crear una enumeración y compilar un diccionario con sus nombres y valores:

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Introducción a los genéricos
- Clases genéricas
- new (restricción)

Clases genéricas (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

Las clases genéricas encapsulan operaciones que no son específicas de un tipo de datos determinado. El uso más común de las clases genéricas es con colecciones como listas vinculadas, tablas hash, pilas, colas y árboles, entre otros. Las operaciones como la adición y eliminación de elementos de la colección se realizan básicamente de la misma manera independientemente del tipo de datos que se almacenan.

Para la mayoría de los escenarios que necesitan clases de colección, el enfoque recomendado es usar las que se proporcionan en la biblioteca de clases .NET. Para más información sobre el uso de estas clases, vea Colecciones genéricas en .NET.

Normalmente, crea clases genéricas empezando con una clase concreta existente, y cambiando tipos en parámetros de tipo de uno en uno hasta que alcanza el equilibrio óptimo de generalización y facilidad de uso. Al crear sus propias clases genéricas, entre las consideraciones importantes se incluyen las siguientes:

- Los tipos que se van a generalizar en parámetros de tipo.

  Como norma, cuantos más tipos pueda parametrizar, más flexible y reutilizable será su código. En cambio, demasiada generalización puede crear código que sea difícil de leer o entender para otros desarrolladores.

- Las restricciones, si existen, que se van a aplicar a los parámetros de tipo (Vea Restricciones de parámetros de tipo).

  Una buena norma es aplicar el máximo número de restricciones posible que todavía le permitan tratar los tipos que debe controlar. Por ejemplo, si sabe que su clase genérica está diseñada para usarse solo con tipos de referencia, aplique la restricción de clase. Esto evitará el uso no previsto de su clase con tipos de valor, y le permitirá usar el operador `as` en `T`, y comprobar si hay valores NULL.

- Si separar el comportamiento genérico en clases base y subclases.

  Como las clases genéricas pueden servir como clases base, las mismas consideraciones de diseño se aplican aquí con clases no genéricas. Vea las reglas sobre cómo heredar de clases base genéricas posteriormente en este tema.

- Si implementar una o más interfaces genéricas.

  Por ejemplo, si está diseñando una clase que se usará para crear elementos en una colección basada en genéricos, puede que tenga que implementar una interfaz como IComparable<T> donde `T` es el tipo de su clase.

Para obtener un ejemplo de una clase genérica simple, vea Introducción a los genéricos.

Las reglas para los parámetros de tipo y las restricciones tienen varias implicaciones para el comportamiento de clase genérico, especialmente respecto a la herencia y a la accesibilidad de miembros. Antes de continuar, debe entender algunos términos. Para una clase genérica `Node<T>,` , el código de cliente puede hacer referencia a la clase especificando un argumento de tipo, para crear un tipo construido cerrado (`Node<int>`). De manera alternativa, puede dejar el parámetro de tipo sin especificar, por ejemplo cuando especifica una clase base genérica, para crear un tipo construido abierto (`Node<T>`). Las clases genéricas pueden heredar de determinadas clases base construidas abiertas o construidas cerradas:

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Las clases no genéricas, en otras palabras, concretas, pueden heredar de clases base construidas cerradas, pero no desde clases construidas abiertas ni desde parámetros de tipo porque no hay ninguna manera en tiempo de ejecución para que el código de cliente proporcione el argumento de tipo necesario para crear instancias de la clase base.

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Las clases genéricas que heredan de tipos construidos abiertos deben proporcionar argumentos de tipo para cualquier parámetro de tipo de clase base que no se comparta mediante la clase heredada, como se demuestra en el código siguiente:

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Las clases genéricas que heredan de tipos construidos abiertos deben especificar restricciones que son un superconjunto de las restricciones del tipo base, o que las implican:

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }
```

Los tipos genéricos pueden usar varios parámetros de tipo y restricciones, de la manera siguiente:

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Tipos construidos cerrados y construidos abiertos pueden usarse como parámetros de método:

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

Si una clase genérica implementa una interfaz, todas las instancias de esa clase se pueden convertir en esa interfaz.

Las clases genéricas son invariables. En otras palabras, si un parámetro de entrada especifica un `List<BaseClass>`, obtendrá un error en tiempo de compilación si intenta proporcionar un `List<DerivedClass>`.

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Genéricos
- Guardar el estado de los enumeradores
- Un puzle de herencia, parte uno

Interfaces genéricas (Guía de programación de C#)

18/03/2020 • 6 minutes to read • Edit Online

A menudo es útil definir interfaces para las clases de colección genéricas o para las clases genéricas que representan los elementos de la colección. Lo preferible para las clases genéricas es usar interfaces genéricas, como IComparable<T> en lugar de IComparable, para evitar las operaciones de conversión boxing y unboxing en los tipos de valor. La biblioteca de clases .NET Framework define varias interfaces genéricas para usarlas con las clases de colección del espacio de nombres System.Collections.Generic.

Cuando una interfaz se especifica como restricción en un parámetro de tipo, solo se pueden usar los tipos que implementan la interfaz. El ejemplo de código siguiente muestra una clase `SortedList<T>` derivada de la clase `GenericList<T>`. Para obtener más información, vea Introducción a los genéricos. `SortedList<T>` agrega la restricción `where T : IComparable<T>`. Esto permite al método `BubbleSort` de `SortedList<T>` usar el método CompareTo genérico con los elementos de lista. En este ejemplo, los elementos de lista son una clase simple, `Person`, que implementa `IComparable<Person>`.

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList()  //constructor
    {
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }
```

```
    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IEnumerable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
```

```
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }

    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

class Program
{
    static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 45, 19, 28, 23, 18, 9, 108, 72, 30, 35 };

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
```

```
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
    }
}
```

Se pueden especificar varias interfaces como restricciones en un solo tipo, de la siguiente manera:

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

Una interfaz puede definir más de un parámetro de tipo, de la siguiente manera:

```
interface IDictionary<K, V>
{
}
```

Las reglas de herencia que se aplican a las clases también se aplican a las interfaces:

```
interface IMonth<T> { }

interface IJanuary     : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T>    : IMonth<T> { }    //No error
//interface IApril<T>  : IMonth<T, U> {}  //Error
```

Las interfaces genéricas pueden heredar de interfaces no genéricas si son contravariantes, lo que significa que solo usan su parámetro de tipo como valor devuelto. En la biblioteca de clases .NET Framework, IEnumerable<T> hereda de IEnumerable porque IEnumerable<T> solo usa `T` en el valor devuelto de GetEnumerator y en el captador de propiedad Current.

Las clases concretas pueden implementar interfaces construidas cerradas, de la siguiente manera:

```
interface IBaseInterface<T> { }

class SampleClass : IBaseInterface<string> { }
```

Las clases genéricas pueden implementar interfaces genéricas o interfaces construidas cerradas siempre que la lista de parámetros de la clase suministre todos los argumentos que necesita la interfaz, de la siguiente manera:

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

Las reglas que controlan la sobrecarga de métodos son las mismas para los métodos incluidos en las clases

genéricas, los structs genéricos o las interfaces genéricas. Para obtener más información, vea Métodos genéricos.

Vea también

- Guía de programación de C#
- Introducción a los genéricos
- interface
- Genéricos

Métodos genéricos (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Un método genérico es un método que se declara con parámetros de tipo, de la manera siguiente:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

En el siguiente ejemplo de código se muestra una manera de llamar al método con `int` para el argumento de tipo:

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

También puede omitir el argumento de tipo y el compilador lo deducirá. La siguiente llamada a `Swap` es equivalente a la llamada anterior:

```
Swap(ref a, ref b);
```

Las mismas reglas para la inferencia de tipos se aplican a los métodos estáticos y a los métodos de instancia. El compilador puede deducir los parámetros de tipo basados en los argumentos de método que se pasan; no puede deducir los parámetros de tipo solo desde un valor devuelto o una restricción. Por lo tanto, la inferencia de tipos no funciona con métodos que no tienen parámetros. La inferencia de tipos se produce en tiempo de compilación antes de que el compilador intente resolver las firmas de método sobrecargadas. El compilador aplica la lógica de inferencia de tipos a todos los métodos genéricos que comparten el mismo nombre. En el paso de resolución de sobrecarga, el compilador incluye solo esos métodos genéricos en los que la inferencia de tipos se ha realizado correctamente.

Dentro de una clase genérica, los métodos no genéricos pueden tener acceso a los parámetros de tipo de nivel de clase, de la manera siguiente:

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

Si define un método genérico que toma los mismos parámetros de tipo que la clase contenedora, el compilador genera la advertencia CS0693 porque, dentro del ámbito del método, el argumento que se ha proporcionado para el `T` interno oculta el argumento que se ha proporcionado para el `T` externo. Si necesita la flexibilidad de llamar a un método de la clase genérica con argumentos de tipo diferentes de los que se han proporcionado cuando se

ha creado una instancia de la clase, considere la posibilidad de proporcionar otro identificador para el parámetro de tipo del método, como se muestra en `GenericList2<T>` en el ejemplo siguiente.

```
class GenericList<T>
{
    // CS0693
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    //No warning
    void SampleMethod<U>() { }
}
```

Use restricciones para permitir operaciones más especializadas en parámetros de tipo de métodos. Esta versión de `Swap<T>`, ahora denominada `SwapIfGreater<T>`, solo puede usarse con argumentos de tipo que implementan IComparable<T>.

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

Los métodos genéricos pueden sobrecargarse en varios parámetros de tipo. Por ejemplo, todos los métodos siguientes pueden ubicarse en la misma clase:

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#.

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Introducción a los genéricos
- Métodos

Genéricos y matrices (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

En C# 2.0 y versiones posteriores, las matrices unidimensionales que tienen un límite inferior de cero implementan IList<T> automáticamente. Esto le permite crear métodos genéricos que pueden usar el mismo código para recorrer en iteración matrices y otros tipos de colección. Esta técnica es útil principalmente para leer datos en colecciones. La interfaz IList<T> no puede usarse para agregar o quitar elementos de una matriz. Se generará una excepción si intenta llamar a un método IList<T> como RemoveAt en una matriz en este contexto.

En el siguiente ejemplo de código se muestra cómo un método genérico único que toma un parámetro de entrada IList<T> puede recorrer en iteración una lista y una matriz, en este caso una matriz de enteros.

```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Genéricos
- Matrices
- Genéricos

Delegados genéricos (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Un delegado puede definir sus propios parámetros de tipo. El código que hace referencia al delegado genérico puede especificar el tipo de argumento para crear un tipo construido abierto, igual que al crear una instancia de una clase genérica o al llamar a un método genérico, como se muestra en el siguiente ejemplo:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# 2.0 tiene una nueva característica denominada conversión de grupo de métodos, que se aplica a los tipos delegados concretos y genéricos, y permite escribir la línea anterior con esta sintaxis simplificada:

```
Del<int> m2 = Notify;
```

Los delegados definidos dentro de una clase genérica pueden usar los parámetros de tipo de la clase genérica de la misma manera que lo hacen los métodos de clase.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

El código que hace referencia al delegado debe especificar el argumento de tipo de la clase contenedora, de la siguiente manera:

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Los delegados genéricos son especialmente útiles para definir eventos basados en el patrón de diseño habitual porque el argumento del remitente puede estar fuertemente tipado y ya no tiene que convertirse a y de Object.

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs> stackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        stackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.stackEvent += o.HandleStackChange;
}
```

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Introducción a los genéricos
- Métodos genéricos
- Clases genéricas
- Interfaces genéricas
- Delegados
- Genéricos

Diferencias entre plantillas de C++ y tipos genéricos de C# (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

Los tipos genéricos de C# y las plantillas de C++ son dos características de lenguaje que ofrecen compatibilidad con tipos parametrizados. Pero existen muchas diferencias entre ambos. En el nivel de sintaxis, los tipos genéricos de C# resultan un enfoque más sencillo que los tipos parametrizados sin la complejidad de las plantillas de C++. Además, C# no intenta ofrecer toda la funcionalidad que ofrecen las plantillas de C++. En el nivel de implementación, la principal diferencia es que las sustituciones de tipo genérico de C# se realizan en tiempo de ejecución y, por tanto, se conserva la información de tipo genérico para los objetos con instancias. Para obtener más información, vea Generics in the Run Time (Genéricos en el tiempo de ejecución).

Estas son las principales diferencias entre plantillas de C++ y tipos genéricos de C#:

- Los tipos genéricos de C# no ofrecen tanta flexibilidad como las plantillas de C++. Por ejemplo, no es posible llamar a operadores aritméticos en una clase de tipos genéricos de C#, aunque es posible llamar a operadores definidos por el usuario.
- C# no permite parámetros de plantilla sin tipo, como `template C<int i> {}`.
- C# no admite la especialización explícita; es decir, una implementación personalizada de una plantilla para un tipo específico.
- C# no admite la especialización parcial: una implementación personalizada de un subconjunto de los argumentos de tipo.
- C# no permite que el parámetro de tipo se use como clase base del tipo genérico.
- C# no permite que los parámetros de tipo tengan tipos predeterminados.
- En C#, un parámetro de tipo genérico no puede ser un tipo genérico, aunque se pueden usar tipos construidos como genéricos. C++ permite parámetros de plantilla.
- C++ permite código que podría no ser válido para todos los parámetros de tipo en la plantilla, que luego busca el tipo específico que se usa como parámetro de tipo. C# requiere que el código de una clase se escriba de manera que funcione con cualquier tipo que cumpla las restricciones. Por ejemplo, en C++ es posible escribir una función que use los operadores aritméticos `+` y `-` en objetos del parámetro de tipo, lo que producirá un error en el momento de creación de instancias de la plantilla con un tipo que no admita estos operadores. C# no permite esto; las únicas construcciones de lenguaje permitidas son las que se pueden deducir de las restricciones.

Vea también

- Guía de programación de C#
- Introducción a los genéricos
- Templates (Plantillas [C++])

Genéricos en el motor en tiempo de ejecución (Guía de programación de C#)

18/03/2020 • 6 minutes to read • Edit Online

Cuando se compila un tipo o método genérico en el lenguaje intermedio de Microsoft (MSIL), contiene metadatos que lo identifican como poseedor de parámetros de tipo. La forma en que se usa MSIL para un tipo genérico depende de si el parámetro de tipo proporcionado es un tipo de valor o de referencia.

Cuando se construye por primera vez un tipo genérico con un tipo de valor como parámetro, el motor de ejecución crea un tipo genérico especializado sustituyendo el parámetro o los parámetros proporcionados en los lugares adecuados del MSIL. Los tipos genéricos especializados se crean una vez para cada tipo de valor único que se usa como parámetro.

Por ejemplo, suponga que el código de su programa ha declarado una pila compuesta por enteros:

```
Stack<int> stack;
```

En este momento, el motor de ejecución genera una versión especializada de la clase Stack<T> sustituyendo el entero adecuadamente para su parámetro. Ahora, cada vez que el código de su programa use una pila de enteros, el motor en tiempo de ejecución vuelve a usar la clase especializada generada Stack<T>. En el ejemplo siguiente, se crean dos instancias de una pila de enteros y comparten una instancia única del código `Stack<int>`:

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

En cambio, suponga que otra clase Stack<T> con un tipo de valor diferente, como un `long` o una estructura definida por el usuario como parámetro, se crea en otro punto del código. Como resultado, el motor de ejecución genera otra versión del tipo genérico y sustituye un `long` en los lugares apropiados en el MSIL. Las conversiones ya no son necesarias porque cada clase genérica especializada contiene de forma nativa el tipo de valor.

Los genéricos funcionan de forma ligeramente distinta para los tipos de referencia. Cuando se construye un tipo genérico por primera vez con un tipo de referencia, el motor en tiempo de ejecución crea un tipo genérico especializado sustituyendo las referencias a objetos para los parámetros del MSIL. Después, cada vez que se crea una instancia de un tipo construido con un tipo de referencia como parámetro, independientemente del tipo que sea, el motor de ejecución vuelve a usar la versión especializada del tipo genérico previamente creada. Esto es posible porque todas las referencias son del mismo tamaño.

Por ejemplo, suponga que tiene dos tipos de referencia, una clase `Customer` y una clase `Order`, y que ha creado una pila de tipos `Customer`:

```
class Customer { }
class Order { }
```

```
Stack<Customer> customers;
```

En este punto, el motor de ejecución genera una versión especializada de la clase Stack<T> que, en lugar de almacenar los datos, almacena referencias a objetos que se rellenarán más tarde. Suponga que la línea siguiente

de código crea una pila de otro tipo de referencia, que se denomina `Order`:

```
Stack<Order> orders = new Stack<Order>();
```

A diferencia de lo que sucede con los tipos de valor, no se crea otra versión especializada de la clase Stack<T> para el tipo `Order`. En su lugar, se crea una instancia de la versión especializada de la clase Stack<T> y se establece la variable `orders` para hacer referencia a ella. Suponga que encuentra una línea de código para crear una pila de tipo `Customer`:

```
customers = new Stack<Customer>();
```

Como con el uso anterior de la clase Stack<T> creada usando el tipo `Order`, se crea otra instancia de la clase especializada Stack<T>. Los punteros contenidos allí se establecen para hacer referencia a un área de memoria del tamaño de un tipo `Customer`. Dado que el número de tipos de referencia puede variar significativamente de un programa a otro, la implementación de genéricos de C# reduce significativamente la cantidad de código limitando a uno el número de clases especializadas creadas por el compilador para las clases genéricas de tipos de referencia.

Además, cuando se crea una instancia de una clase de C# genérica mediante un parámetro de tipo de valor o de referencia se puede consultar en tiempo de ejecución mediante reflexión, y se puede comprobar tanto su tipo real como su parámetro de tipo.

Vea también

- System.Collections.Generic
- Guía de programación de C#
- Introducción a los genéricos
- Genéricos

Genéricos y reflexión (Guía de programación de C#)

29/06/2020 • 5 minutes to read • Edit Online

Dado que Common Language Runtime (CLR) tiene acceso a la información de tipos genéricos en tiempo de ejecución, se puede usar la reflexión para obtener información sobre los tipos genéricos de la misma manera que para los tipos no genéricos. Para obtener más información, vea Genéricos en el motor en tiempo de ejecución.

En .NET Framework 2.0 se agregan nuevos miembros a la clase Type para habilitar la información de tiempo de ejecución para tipos genéricos. Vea la documentación sobre estas clases para obtener más información sobre cómo usar estos métodos y propiedades. El espacio de nombres System.Reflection.Emit también contiene los miembros nuevos que admiten genéricos. Vea Cómo: Definir un tipo genérico con emisión de reflexión.

Para obtener una lista de las condiciones invariables para los términos usados en la reflexión genérica, vea los comentarios de la propiedad IsGenericType.

NOMBRE DE MIEMBRO SYSTEM.TYPE	DESCRIPCIÓN
IsGenericType	Devuelve true si un tipo es genérico.
GetGenericArguments	Devuelve una matriz de objetos `Type` que representan los argumentos de tipo proporcionados para un tipo construido, o los parámetros de tipo de una definición de tipo genérico.
GetGenericTypeDefinition	Devuelve la definición de tipo genérico subyacente para el tipo construido actual.
GetGenericParameterConstraints	Devuelve una matriz de objetos `Type` que representan las restricciones en el parámetro de tipo genérico actual.
ContainsGenericParameters	Devuelve true si el tipo o cualquiera de sus tipos o métodos envolventes contiene los parámetros de tipo para los que no se proporcionaron tipos específicos.
GenericParameterAttributes	Obtiene una combinación de marcas `GenericParameterAttributes` que describen las restricciones especiales del parámetro de tipo genérico actual.
GenericParameterPosition	Para un objeto `Type` que representa un parámetro de tipo, obtiene la posición del parámetro de tipo en la lista de parámetros de tipo de la definición de tipo genérico o de método genérico que declaró el parámetro de tipo.
IsGenericParameter	Obtiene un valor que indica si el objeto `Type` actual representa un parámetro de tipo de una definición de un tipo o método genérico.
IsGenericTypeDefinition	Obtiene un valor que indica si el objeto Type actual representa una definición de tipo genérico, a partir de la cual se pueden construir otros tipos genéricos. Devuelve true si el tipo representa la definición de un tipo genérico.

NOMBRE DE MIEMBRO SYSTEM.TYPE	DESCRIPCIÓN
DeclaringMethod	Devuelve el método genérico que definió el parámetro de tipo genérico actual o NULL si el parámetro de tipo no se definió mediante un método genérico.
MakeGenericType	Sustituye los elementos de una matriz de tipos por los parámetros de tipo de la definición de tipo genérico actual y devuelve un objeto Type que representa el tipo construido resultante.

Además, los miembros de la clase MethodInfo habilitan la información en tiempo de ejecución para métodos genéricos. Para obtener una lista de las condiciones invariables para los términos usados para reflejarse en métodos genéricos, vea los comentarios de la propiedad IsGenericMethod.

NOMBRE DE MIEMBRO SYSTEM.REFLECTION.MEMBERINFO	DESCRIPCIÓN
IsGenericMethod	Devuelve true si un método es genérico.
GetGenericArguments	Devuelve una matriz de objetos Type que representan los argumentos de tipo de un método genérico construido o los parámetros de tipo de una definición de método genérico.
GetGenericMethodDefinition	Devuelve la definición de método genérico subyacente para el método construido actual.
ContainsGenericParameters	Devuelve true si el método o cualquiera de sus tipos envolventes contiene los parámetros de tipo para los que no se proporcionaron tipos específicos.
IsGenericMethodDefinition	Devuelve True si el tipo MethodInfo actual representa la definición de un método genérico.
MakeGenericMethod	Sustituye los elementos de una matriz de tipos por los parámetros de tipo de la definición de método genérico actual y devuelve un objeto MethodInfo que representa el método construido resultante.

Vea también

- Guía de programación de C#
- Genéricos
- Reflexión y tipos genéricos
- Genéricos

Genéricos y atributos (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Los atributos pueden aplicarse a los tipos genéricos de la misma manera que los tipos no genéricos. Para obtener más información sobre la aplicación de los atributos, vea Atributos.

Los atributos personalizados solo se permiten para hacer referencia a tipos genéricos abiertos, que son tipos genéricos para los que no se proporciona ningún argumento de tipo, y tipos genéricos construidos cerrados, que proporcionan argumentos para todos los parámetros de tipo.

En los ejemplos siguientes se usa este atributo personalizado:

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

Un atributo puede hacer referencia a un tipo genérico abierto:

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Especifica varios parámetros de tipo con el número de comas apropiado. En este ejemplo, `GenericClass2` tiene dos parámetros de tipo:

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

Un atributo puede hacer referencia a un tipo genérico construido cerrado:

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

Un atributo que hace referencia a un parámetro de tipo genérico provocará un error en tiempo de compilación:

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error
class ClassD<T> { }
```

Un tipo genérico no puede heredarse de Attribute:

```
//public class CustomAtt<T> : System.Attribute {}  //Error
```

Para obtener información sobre un tipo genérico o un parámetro de tipo en tiempo de ejecución, puede usar los métodos de System.Reflection. Para obtener más información, vea Genéricos y reflexión.

Vea también

- Guía de programación de C#
- Genéricos
- Atributos

Espacios de nombres (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Los espacios de nombres se usan mucho en programación de C# de dos maneras. En primer lugar, .NET usa espacios de nombres para organizar sus clases de la siguiente manera:

```
System.Console.WriteLine("Hello World!");
```

System es un espacio de nombres y Console es una clase de ese espacio de nombres. La palabra clave `using` se puede usar para que no se necesite el nombre completo, como en el ejemplo siguiente:

```
using System;
```

```
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

Para más información, vea using (Directiva).

En segundo lugar, declarar sus propios espacios de nombres puede ayudarle a controlar el ámbito de nombres de clase y método en proyectos de programación grandes. Use la palabra clave namespace para declarar un espacio de nombres, como en el ejemplo siguiente:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

El nombre del espacio de nombres debe ser un nombre de identificador de C# válido.

Información general sobre los espacios de nombres

Los espacios de nombres tienen las propiedades siguientes:

- Organizan proyectos de código de gran tamaño.
- Se delimitan mediante el operador `.`.
- La directiva `using` obvia la necesidad de especificar el nombre del espacio de nombres para cada clase.
- El espacio de nombres `global` es el espacio de nombres "raíz": `global::System` siempre hará referencia al espacio de nombres System de .NET.

especificación del lenguaje C#

Para más información, vea la sección Espacio de nombres de la Especificación del lenguaje C#.

Vea también

- Guía de programación de C#
- Utilizar espacios de nombres
- Procedimiento para usar el espacio de nombres My
- Nombres de identificador
- using (directiva)
- Operador ::

Uso de espacios de nombres (Guía de programación de C#)

29/06/2020 • 7 minutes to read • Edit Online

Los espacios de nombres se usan mucho en programas de C# de dos maneras. En primer lugar, las clases de .NET usan espacios de nombres para organizar sus clases. En segundo lugar, declarar sus propios espacios de nombres puede ayudar a controlar el ámbito de nombres de clase y método en proyectos de programación grandes.

Acceso a los espacios de nombres

La mayoría de las aplicaciones de C# comienzan con una sección de directivas `using`. En esta sección se muestran los espacios de nombres que la aplicación usará frecuentemente, y evita que el programador especifique un nombre completo cada vez que se use un método que se incluye dentro.

Por ejemplo, incluyendo la línea:

```
using System;
```

Al comienzo de un programa, el programador puede usar el código:

```
Console.WriteLine("Hello, World!");
```

En lugar de:

```
System.Console.WriteLine("Hello, World!");
```

Alias de espacios de nombres

También puede usar la directiva `using` para crear un alias de un espacio de nombres. Use el calificador de alias de espacio de nombres `::` para tener acceso a los miembros del espacio de nombres con alias. En el ejemplo siguiente se muestra cómo crear y usar un alias para un espacio de nombres:

```
using generics = System.Collections.Generic;

namespace AliasExample
{
    class TestClass
    {
        static void Main()
        {
            generics::Dictionary<string, int> dict = new generics::Dictionary<string, int>()
            {
                ["A"] = 1,
                ["B"] = 2,
                ["C"] = 3
            };

            foreach (string name in dict.Keys)
            {
                System.Console.WriteLine($"{name} {dict[name]}");
            }
            // Output:
            // A 1
            // B 2
            // C 3
        }
    }
}
```

Uso de los espacios de nombres para controlar el ámbito

La palabra clave `namespace` se usa para declarar un ámbito. La capacidad de crear ámbitos en su proyecto le ayuda a organizar código y le permite crear tipos únicos globalmente. En el ejemplo siguiente, una clase denominada `SampleClass` se define en dos espacios de nombres, uno anidado dentro de otro. El token `.` se usa para diferenciar a qué método se llama.

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }

    // Create a nested namespace, and define another class.
    namespace NestedNamespace
    {
        class SampleClass
        {
            public void SampleMethod()
            {
                System.Console.WriteLine(
                  "SampleMethod inside NestedNamespace");
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Displays "SampleMethod inside SampleNamespace."
            SampleClass outer = new SampleClass();
            outer.SampleMethod();

            // Displays "SampleMethod inside SampleNamespace."
            SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
            outer2.SampleMethod();

            // Displays "SampleMethod inside NestedNamespace."
            NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
            inner.SampleMethod();
        }
    }
}
```

Nombres completos

Los espacios de nombres y los tipos tienen títulos únicos descritos mediante nombres completos que indican una jerarquía lógica. Por ejemplo, la instrucción `A.B` implica que `A` es el nombre del espacio de nombres o el tipo, y `B` está anidado en su interior.

En el ejemplo siguiente, existen espacios de nombres y clases anidadas. El nombre completo se indica como un comentario que sigue a cada entidad.

```
namespace N1     // N1
{
    class C1      // N1.C1
    {
        class C2   // N1.C1.C2
        {
        }
    }
    namespace N2  // N1.N2
    {
        class C2   // N1.N2.C2
        {
        }
    }
}
```

En el segmento de código anterior:

- El espacio de nombres `N1` es un miembro del espacio de nombres global. Su nombre completo es `N1`.
- El espacio de nombres `N2` es un miembro de `N1`. Su nombre completo es `N1.N2`.
- La clase `C1` es un miembro de `N1`. Su nombre completo es `N1.C1`.
- El nombre de la clase `C2` se usa dos veces en este código. En cambio, los nombres completos son únicos. La primera instancia de `C2` se declara dentro de `C1`; por lo tanto, su nombre completo es: `N1.C1.C2`. La segunda instancia de `C2` se declara dentro de un espacio de nombres `N2`; por lo tanto, su nombre completo es `N1.N2.C2`.

Con el segmento de código anterior, puede agregar un nuevo miembro de clase, `C3`, al espacio de nombres `N1.N2` de la manera siguiente:

```
namespace N1.N2
{
    class C3   // N1.N2.C3
    {
    }
}
```

En general, use el calificador de alias de espacio de nombres `::` para hacer referencia a un alias de espacio de nombres o `global::` para hacer referencia al espacio de nombres global y `.` para calificar tipos o miembros.

Es un error usar `::` con un alias que haga referencia a un tipo en lugar de a un espacio de nombres. Por ejemplo:

```
using Alias = System.Console;
```

```
class TestClass
{
    static void Main()
    {
        // Error
        //Alias::WriteLine("Hi");

        // OK
        Alias.WriteLine("Hi");
    }
}
```

Recuerde que la palabra `global` no es un alias predefinido; por lo tanto, `global.X` no tiene ningún significado especial. Adquiere un significado especial solo cuando se usa con `::`.

La advertencia del compilador CS0440 se genera si define un alias denominado global porque `global::` siempre hace referencia al espacio de nombres global y no a un alias. Por ejemplo, la línea siguiente genera la advertencia:

```
using global = System.Collections;   // Warning
```

Usar `::` con alias es una buena idea y protege contra la introducción inesperada de tipos adicionales. Por ejemplo, considere este ejemplo:

```
using Alias = System;
```

```
namespace Library
{
    public class C : Alias::Exception { }
}
```

Esto funciona, pero si un tipo denominado `Alias` fuera a introducirse posteriormente, `Alias.` se enlazaría a ese tipo en su lugar. Usar `Alias::Exception` garantiza que `Alias` se trata como un alias de espacio de nombres y no se confunde con un tipo.

Vea también

- Guía de programación de C#
- Espacios de nombres
- Expresión de acceso a miembro
- Operador ::
- extern alias

Procedimiento Utilizar el espacio de nombres My (Guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

El espacio de nombres Microsoft.VisualBasic.MyServices (`My` en Visual Basic) proporciona acceso fácil e intuitivo a una serie de clases de .NET, lo que le permite escribir código que interactúe con el equipo, la aplicación, la configuración, los recursos, etc. Aunque se diseñó originalmente para usarse con Visual Basic, el espacio de nombres `MyServices` puede usarse en aplicaciones de C#.

Para obtener más información sobre el uso del espacio de nombres `MyServices` de Visual Basic, consulte Desarrollo con My.

Agregar una referencia

Para poder usar las clases `MyServices` en la solución, debe agregar una referencia a la biblioteca de Visual Basic.

Cómo agregar una referencia a la biblioteca de Visual Basic

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el nodo **Referencias** y seleccione **Agregar referencia**.

2. Cuando aparezca el cuadro de diálogo **Referencias**, desplácese hacia abajo en la lista y seleccione Microsoft.VisualBasic.dll.

   También es posible que quiera incluir la siguiente línea en la sección `using` al principio del programa.

   ```
   using Microsoft.VisualBasic.Devices;
   ```

Ejemplo

En este ejemplo, se llama a distintos métodos estáticos contenidos en el espacio de nombres `MyServices`. Para que pueda compilar este código, debe agregarse al proyecto una referencia a Microsoft.VisualBasic.DLL.

```
using System;
using Microsoft.VisualBasic.Devices;

class TestMyServices
{
    static void Main()
    {
        // Play a sound with the Audio class:
        Audio myAudio = new Audio();
        Console.WriteLine("Playing sound...");
        myAudio.Play(@"c:\WINDOWS\Media\chimes.wav");

        // Display time information with the Clock class:
        Clock myClock = new Clock();
        Console.Write("Current day of the week: ");
        Console.WriteLine(myClock.LocalTime.DayOfWeek);
        Console.Write("Current date and time: ");
        Console.WriteLine(myClock.LocalTime);

        // Display machine information with the Computer class:
        Computer myComputer = new Computer();
        Console.WriteLine("Computer name: " + myComputer.Name);

        if (myComputer.Network.IsAvailable)
        {
            Console.WriteLine("Computer is connected to network.");
        }
        else
        {
            Console.WriteLine("Computer is not connected to network.");
        }
    }
}
```

No se puede llamar a todas las clases en el espacio de nombres `MyServices` desde una aplicación de C#: por ejemplo, la clase FileSystemProxy no es compatible. En este caso concreto, los métodos estáticos que forman parte de FileSystem, que también se incluyen en el archivo VisualBasic.dll, pueden usarse en su lugar. Por ejemplo, aquí se muestra cómo usar uno de estos métodos para duplicar un directorio:

```
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

Vea también

- Guía de programación de C#
- Espacios de nombres
- Utilizar espacios de nombres

Código no seguro y punteros (guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Para mantener la seguridad de tipos, C# no admite la aritmética de puntero de forma predeterminada. En cambio, mediante el uso de la palabra clave unsafe, puede definir un contexto no seguro en el que se pueden usar punteros. Para más información sobre los punteros, vea Tipos de puntero.

> **NOTE**
>
> En Common Language Runtime (CLR), el código no seguro se conoce como código no comprobable. El código no seguro en C# no es necesariamente peligroso; solo es código cuya seguridad no puede comprobar CLR. Por lo tanto, CLR solo ejecutará código no seguro si se encuentra en un ensamblado de plena confianza. Si usa código no seguro, es su responsabilidad asegurarse de que el código no presenta riesgos de seguridad o errores de puntero.

Introducción sobre el código no seguro

El código no seguro tiene las propiedades siguientes:

- Los métodos, tipos y bloques de código se pueden definir como no seguros.
- En algunos casos, el código no seguro puede aumentar el rendimiento de la aplicación al eliminar las comprobaciones de límites de matriz.
- El código no seguro es necesario al llamar a funciones nativas que requieren punteros.
- El código no seguro presenta riesgos para la seguridad y la estabilidad.
- El código que contiene bloques no seguros debe compilarse con la opción del compilador -unsafe.

Secciones relacionadas

Para obtener más información, consulte:

- Tipos de puntero
- Búferes de tamaño fijo

especificación del lenguaje C#

Para más información, vea el tema sobre código no seguro de la Especificación del lenguaje C#.

Vea también

- Guía de programación de C#
- unsafe

Búferes de tamaño fijo (Guía de programación de C#)

29/06/2020 • 5 minutes to read • Edit Online

En C#, puede usar la instrucción fixed para crear un búfer con una matriz de tamaño fijo en una estructura de datos. Los búferes de tamaño fijo resultan útiles al escribir métodos que interoperan con orígenes de datos de otros lenguajes o plataformas. La matriz fija puede tomar cualquiera de los atributos o modificadores permitidos para los miembros de structs normales. La única restricción es que el tipo de matriz debe ser `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float` o `double`.

```
private fixed char name[30];
```

Comentarios

En el código seguro, un struct de C# que contiene una matriz no contiene los elementos de matriz. En su lugar, el struct contiene una referencia a los elementos. Puede insertar una matriz de tamaño fijo en un struct cuando se usa en un bloque de código no seguro.

El tamaño del siguiente `struct` no depende del número de elementos en la matriz, ya que `pathName` es una referencia:

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

Un `struct` puede contener una matriz insertada en el código no seguro. En el siguiente ejemplo, la matriz `fixedBuffer` tiene un tamaño fijo. Se usa una instrucción `fixed` para establecer un puntero al primer elemento. Se accede a los elementos de la matriz mediante este puntero. La instrucción `fixed` ancla el campo de instancia `fixedBuffer` a una ubicación concreta en la memoria.

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

El tamaño de la matriz `char` de 128 elementos es 256 bytes. Los búferes char de tamaño fijo siempre admiten dos bytes por carácter, independientemente de la codificación. Esto es verdadero, incluso cuando se calculan las referencias de los búferes de caracteres a los métodos API o structs con `CharSet = CharSet.Auto` o `CharSet = CharSet.Ansi`. Para obtener más información, vea CharSet.

En el ejemplo anterior se muestra cómo acceder a campos `fixed` sin anclar, lo que está disponible a partir de C# 7.3.

Otra matriz de tamaño fijo común es la matriz bool. Los elementos de una matriz `bool` siempre tienen un byte de tamaño. Las matrices `bool` no son adecuadas para crear matrices de bits o búferes.

Los búferes de tamaño fijo se compilan con el atributo System.Runtime.CompilerServices.UnsafeValueTypeAttribute, que indica a Common Language Runtime (CLR) que un tipo contiene una matriz no administrada que puede provocar un desbordamiento. Esto es similar a la memoria creada mediante stackalloc, que habilita automáticamente las características de detección de saturación del búfer en CLR. En el ejemplo anterior se muestra cómo podría existir un búfer de tamaño fijo en un `unsafe struct`.

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}
```

El código de C# generado por el compilador para `Buffer` se atribuye de la siguiente forma:

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

Los búferes de tamaño fijo son diferentes de las matrices normales en los siguientes puntos:

- Solo se pueden usar en un contexto no seguro.
- Solo pueden ser campos de instancia de structs.
- Siempre son vectores o matrices unidimensionales.
- La declaración debe incluir la longitud, como `fixed char id[8]`. No se puede usar `fixed char id[]`.

Vea también

- Guía de programación de C#
- Código no seguro y punteros
- fixed (instrucción)
- Interoperabilidad

Tipos de puntero (Guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

En un contexto no seguro, un tipo puede ser un tipo de puntero, un tipo de valor o un tipo de referencia. Una declaración de tipos de puntero toma una de las siguientes formas:

```
type* identifier;
void* identifier; //allowed but not recommended
```

El tipo especificado antes de `*` en un tipo de puntero se denomina **tipo referente**. Solo un tipo no administrado puede ser un tipo de referente.

Los tipos de puntero no heredan de object y no existe ninguna conversión entre tipos de puntero y `object`. Además, las conversiones boxing y unboxing no admiten punteros. Sin embargo, puede realizar la conversión entre diferentes tipos de puntero y entre tipos de puntero y tipos enteros.

Cuando declare varios punteros en la misma declaración, únicamente debe escribir el asterisco (*) con el tipo subyacente; no se usa como prefijo en cada nombre de puntero. Por ejemplo:

```
int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

Un puntero no puede señalar a una referencia ni a un struct que contenga referencias, porque una referencia de objeto puede recolectarse como elemento no utilizado aunque haya un puntero que la señale. El recolector de elementos no utilizados no realiza un seguimiento de si algún tipo de puntero señala a un objeto.

El valor de la variable de puntero de tipo `myType*` es la dirección de una variable de tipo `myType`. A continuación se muestran ejemplos de declaraciones de tipos de puntero:

EJEMPLO	DESCRIPCIÓN
`int* p`	`p` es un puntero a un entero.
`int** p`	`p` es un puntero a un puntero a un entero.
`int*[] p`	`p` es una matriz unidimensional de punteros a enteros.
`char* p`	`p` es un puntero a un valor char.
`void* p`	`p` es un puntero a un tipo desconocido.

El operador de direccionamiento indirecto del puntero * puede usarse para obtener acceso al contenido de la ubicación señalada por la variable de puntero. Por ejemplo, consideremos la siguiente declaración:

```
int* myVariable;
```

La expresión `*myVariable` denota la variable `int` que se encuentra en la dirección contenida en `myVariable`.

Hay varios ejemplos de punteros en los temas fixed (Instrucción) y Conversiones de puntero (Guía de

programación de C#). En el ejemplo siguiente se usa la palabra clave `unsafe` y la instrucción `fixed` y se muestra cómo incrementar un puntero interior. Puede pegar este código en la función Main de una aplicación de consola para ejecutarla. Estos ejemplos deben compilarse con el conjunto de opciones del compilador -unsafe.

```
// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
--------
10
11
12
--------
12
*/
```

No se puede aplicar el operador de direccionamiento indirecto a un puntero de tipo `void*`. Sin embargo, es posible usar una conversión para convertir un puntero void en cualquier otro tipo de puntero y viceversa.

Un puntero puede ser `null`. La aplicación del operador de direccionamiento indirecto a un puntero NULL da como resultado un comportamiento definido por la implementación.

Pasar punteros entre métodos puede provocar un comportamiento no definido. Valore la posibilidad de usar un método que devuelva un puntero a una variable local mediante un parámetro `in`, `out` o `ref`, o bien como resultado de la función. Si el puntero se estableció en un bloque fijo, es posible que la variable a la que señala ya no sea fija.

En la tabla siguiente se muestran los operadores e instrucciones que pueden funcionar en punteros en un contexto no seguro:

OPERADOR/INSTRUCCIÓN	USAR
`*`	Realiza el direccionamiento indirecto del puntero.
`->`	Obtiene acceso a un miembro de un struct a través de un puntero.
`[]`	Indiza un puntero.
`&`	Obtiene la dirección de una variable.
`++` y `--`	Incrementa y disminuye los punteros.
`+` y `-`	Realiza aritmética con punteros.
`==`, `!=`, `<`, `>`, `<=` y `>=`	Compara los punteros.
`stackalloc`	Asigna memoria en la pila.
Instrucción `fixed`	Fija provisionalmente una variable para que pueda encontrarse su dirección.

Para obtener más información sobre los operadores relacionados con el puntero, vea Operadores relacionados con el puntero.

Especificación del lenguaje C#

Para obtener más información, vea la sección Tipos de puntero de Especificación del lenguaje C#.

Vea también

- Guía de programación de C#
- Código no seguro y punteros
- Conversiones de puntero
- Tipos de referencia
- Tipos de valor
- unsafe

Conversiones de puntero (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

En la tabla siguiente se muestran las conversiones de puntero implícitas predefinidas. Las conversiones implícitas pueden ocurrir en muchas situaciones, incluidas las instrucciones de asignación y de invocación de método.

Conversiones de puntero implícitas

DE	PARA
Cualquier tipo de puntero	void*
null	Cualquier tipo de puntero

La conversión de puntero explícita se usa para realizar conversiones, donde no existe ninguna conversión implícita, mediante una expresión de conversión. En la siguiente tabla se muestran estas conversiones.

Conversiones de puntero explícitas

DE	PARA
Cualquier tipo de puntero	Cualquier otro tipo de puntero
sbyte, byte, short, ushort, int, uint, long o ulong	Cualquier tipo de puntero
Cualquier tipo de puntero	sbyte, byte, short, ushort, int, uint, long o ulong

Ejemplo

En el ejemplo siguiente, un puntero a `int` se convierte en un puntero a `byte`. Tenga en cuenta que el puntero señala al byte dirigido más bajo de la variable. Cuando incrementa el resultado sucesivamente, hasta el tamaño de `int` (4 bytes), puede mostrar los bytes restantes de la variable.

```
// compile with: -unsafe
```

```
class ClassConvert
{
    static void Main()
    {
        int number = 1024;

        unsafe
        {
            // Convert to byte:
            byte* p = (byte*)&number;

            System.Console.Write("The 4 bytes of the integer:");

            // Display the 4 bytes of the int variable:
            for (int i = 0 ; i < sizeof(int) ; ++i)
            {
                System.Console.Write(" {0:X2}", *p);
                // Increment the pointer:
                p++;
            }
            System.Console.WriteLine();
            System.Console.WriteLine("The value of the integer: {0}", number);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
```

Vea también

- Guía de programación de C#
- Tipos de puntero
- Tipos de referencia
- Tipos de valor
- unsafe
- fixed (instrucción)
- stackalloc

Procedimiento Utilizar punteros para copiar una matriz de bytes (Guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

En el ejemplo siguiente se usan punteros para copiar bytes de una matriz a otra.

En este ejemplo se usa la palabra clave unsafe, que permite el uso de punteros en el método `Copy`. La instrucción fixed se usa para declarar punteros a las matrices de origen y destino. La instrucción `fixed` *ancla* la ubicación de las matrices de origen y destino en memoria, para que no puedan ser desplazadas por la recolección de elementos no utilizados. Estos bloques de memoria para las matrices se desanclan cuando finaliza el bloque `fixed`. Dado que el método `Copy` de este ejemplo usa la palabra clave `unsafe`, se debe compilar con la opción del compilador -unsafe.

Este ejemplo accede a los elementos de ambas matrices con índices en lugar de con un segundo puntero no administrado. La declaración de los punteros `pSource` y `pTarget` ancla las matrices. Esta característica está disponible a partir de C# 7.3.

Ejemplo

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}

static void UnsafeCopyArrays()
{
```

```
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}
```

Vea también

- Guía de programación de C#
- Código no seguro y punteros
- -unsafe (Opciones del compilador de C#)
- Recolección de elementos no utilizados

Comentarios de documentación XML (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

En C#, puede crear documentación para el código incluyendo elementos XML en campos de comentario especiales (se indica con barras diagonales triples) en el código fuente directamente delante del bloque de código al que hacen referencia los comentarios, por ejemplo.

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

Cuando se compila con la opción -doc, el compilador buscará todas las etiquetas XML en el código fuente y creará un archivo de documentación XML. Para crear la documentación final basada en el archivo generado por el compilador, puede crear una herramienta personalizada o usar una herramienta como DocFX o Sandcastle.

Para hacer referencia a elementos XML (por ejemplo, la función procesa los elementos XML concretos que desea describir en un comentario de documentación XML), puede usar el mecanismo de entrecomillado estándar (`<` y `>`). Para hacer referencia a identificadores genéricos en elementos de referencia de código (`cref`), puede usar los caracteres de escape (por ejemplo, `cref="List<T>"`) o llaves (`cref="List{T}"`). Como caso especial, el compilador analiza las llaves como corchetes angulares para que la creación del comentario de documentación resulte menos complicada al hacer referencia a identificadores genéricos.

> **NOTE**
> Los comentarios de documentación XML no son metadatos; no se incluyen en el ensamblado compilado y, por tanto, no se puede obtener acceso a ellos mediante reflexión.

En esta sección

- Etiquetas recomendadas para los comentarios de documentación
- Procesamiento del archivo XML
- Delimitadores para etiquetas de documentación
- Procedimiento para usar las características de la documentación XML

Secciones relacionadas

Para obtener más información, consulte:

- -doc (Procesamiento de comentarios de documentación)

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#

Etiquetas recomendadas para comentarios de documentación (guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

El compilador de C# procesa los comentarios de documentación de su código y les aplica formato como XML en un archivo cuyo nombre especifica en la opción de línea de comandos **/doc**. Para crear la documentación final basada en el archivo generado por el compilador, puede crear una herramienta personalizada o usar una herramienta como DocFX o Sandcastle.

Las etiquetas se procesan en construcciones de código como tipos y miembros de tipo.

> **NOTE**
> Los comentarios de documentación no pueden aplicarse en un espacio de nombres.

El compilador procesará cualquier etiqueta que sea un XML válido. Las siguientes etiquetas proporcionan funciones de uso general en la documentación de usuario.

Etiquetas

<c>	<para>	<see>*	<value>
<code>	<param>*	<seealso>*	
<example>	<paramref>	<summary>	
<exception>*	<permission>*	<typeparam>*	
<include>*	<remarks>	<typeparamref>	
<list>	<inheritdoc>	<returns>	

(* denota que el compilador comprueba la sintaxis).

Si quiere que aparezcan corchetes angulares en el texto de un comentario de documentación, utilice la codificación HTML de `<` y `>`, que es `<` y `>` respectivamente. Esta codificación se muestra en el ejemplo siguiente.

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

Vea también

- Guía de programación de C#
- -doc (opciones del compilador de C#)
- Comentarios de documentación XML

Procesamiento del archivo XML (guía de programación de C#)

29/06/2020 • 9 minutes to read • Edit Online

El compilador genera una cadena de identificador para cada construcción del código que se etiqueta para generar documentación. (Para más información sobre cómo etiquetar el código, vea Etiquetas recomendadas para comentarios de documentación). La cadena de identificador identifica la construcción de forma exclusiva. Los programas que procesan el archivo XML pueden usar la cadena de identificador para identificar el elemento de reflexión o de metadatos correspondiente de .NET al que se aplica la documentación.

Cadenas de identificador

El archivo XML no es una representación jerárquica del código. Es una lista plana que tiene un identificador generado para cada elemento.

El compilador cumple las siguientes reglas cuando genera las cadenas de identificador:

- No hay ningún espacio en blanco en la cadena.
- La primera parte de la cadena identifica el tipo de miembro mediante un carácter único seguido de dos puntos. Se utilizan los siguientes tipos de miembros:

CARÁCTER	TIPO DE MIEMBRO	NOTAS
N	namespace	No puede agregar comentarios de documentación a un espacio de nombres, pero sí puede hacer referencias cruzadas a ellos, en caso de que se admitan.
T	type	Un tipo puede ser una clase, una interfaz, una estructura, una enumeración o un delegado.
F	campo	
P	propiedad	Incluye indizadores u otras propiedades indizadas.
M	método	Incluye métodos especiales, como constructores y operadores.
E	event	
!	cadena de error	El resto de la cadena proporciona información sobre el error. El compilador de C# genera información de error para vínculos que no se puede resolver.

- La segunda parte de la cadena es el nombre completo del elemento, que empieza por la raíz del espacio de nombres. El nombre del elemento, sus tipos envolventes y el espacio de nombres están separados por

puntos. Si el nombre del elemento ya contiene puntos, estos se reemplazan por el signo hash ("#"). Se supone que ningún elemento tiene un signo hash directamente en su nombre. Por ejemplo, el nombre completo del constructor de String es "System.String.#ctor".

- Para las propiedades y los métodos, se indica la lista de parámetros entre paréntesis. Si no hay ningún parámetro, tampoco habrá paréntesis. Los parámetros se separan mediante comas. La codificación de cada parámetro indica directamente cómo se codifica en una firma de .NET:
  - Tipos base. Los tipos habituales (ELEMENT_TYPE_CLASS o ELEMENT_TYPE_VALUETYPE) se representan como el nombre completo del tipo.
  - Los tipos intrínsecos (por ejemplo, ELEMENT_TYPE_I4, ELEMENT_TYPE_OBJECT, ELEMENT_TYPE_STRING, ELEMENT_TYPE_TYPEDBYREF y ELEMENT_TYPE_VOID) se representan como el nombre completo del tipo completo correspondiente. Por ejemplo, System.Int32 o System.TypedReference.
  - ELEMENT_TYPE_PTR se representa como un "*" después del tipo modificado.
  - ELEMENT_TYPE_BYREF se representa como un "@" después del tipo modificado.
  - ELEMENT_TYPE_PINNED se representa como "^" después del tipo modificado. El compilador de C# nunca genera este resultado.
  - ELEMENT_TYPE_CMOD_REQ se representa como "|" y el nombre completo de la clase modificadora, después del tipo modificado. El compilador de C# nunca genera este resultado.
  - ELEMENT_TYPE_CMOD_OPT se representa como "!" y el nombre completo de la clase modificadora, después del tipo modificado.
  - ELEMENT_TYPE_SZARRAY se representa como "[]" después del tipo de elemento de la matriz.
  - ELEMENT_TYPE_GENERICARRAY se representa como "[?]" después del tipo de elemento de la matriz. El compilador de C# nunca genera este resultado.
  - ELEMENT_TYPE_ARRAY se representa como [*límite inferior*:`size`,*límite inferior*:`size`], donde el número de comas es la clasificación - 1, y los límites inferiores y el tamaño de cada dimensión, si se conocen, se representan en decimales. Si no se especifican un límite inferior ni un tamaño, se omiten. Si se omiten el límite inferior y el tamaño de una dimensión determinada, también se omite ":". Por ejemplo, una matriz de dos dimensiones con 1 como los límites inferiores y tamaños no especificados es [1:,1:].
  - ELEMENT_TYPE_FNPTR se representa como "=FUNC:`type`(*signature*)", donde `type` es el tipo de valor devuelto y *signature* se corresponde con los argumentos del método. Si no hay ningún argumento, se omiten los paréntesis. El compilador de C# nunca genera este resultado.

  Los siguientes componentes de la firma no se representan porque nunca se usan para diferenciar métodos sobrecargados:

  - Convención de llamada
  - Tipo de valor devuelto
  - ELEMENT_TYPE_SENTINEL
- Solo para los operadores de conversión (`op_Implicit` y `op_Explicit`), el valor devuelto del método se codifica como "~" seguido por el tipo de valor devuelto.
- Para tipos genéricos, el nombre del tipo está seguido de una tilde aguda y después de un número que indica el número de parámetros de tipo genérico. Por ejemplo:

`<member name="T:SampleClass`2">` es la etiqueta de un tipo que se define como `public class SampleClass<T, U>`.

Para los métodos que toman tipos genéricos como parámetros, los parámetros de tipo genérico se especifican como números precedidos por tildes graves (por ejemplo, `0,`1). Cada número representa una notación de matriz de base cero para los parámetros genéricos del tipo.

Ejemplos

En los ejemplos siguientes, se muestra cómo se generan las cadenas de identificador para una clase y sus miembros:

```
namespace N
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:N.X".
    /// </summary>
    public unsafe class X
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:N.X.#ctor".
        /// </summary>
        public X() { }

        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:N.X.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public X(int i) { }

        /// <summary>
        /// Enter description here for field q.
        /// ID string generated is "F:N.X.q".
        /// </summary>
        public string q;

        /// <summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:N.X.PI".
        /// </summary>
        public const double PI = 3.14;

        /// <summary>
        /// Enter description for method f.
        /// ID string generated is "M:N.X.f".
        /// </summary>
        /// <returns>Describe return value.</returns>
        public int f() { return 1; }

        /// <summary>
        /// Enter description for method bb.
        /// ID string generated is "M:N.X.bb(System.String,System.Int32@,System.Void*)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <param name="y">Describe parameter.</param>
        /// <param name="z">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int bb(string s, ref int y, void* z) { return 1; }

        /// <summary>
        /// Enter description for method gg.
        /// ID string generated is "M:N.X.gg(System.Int16[],System.Int32[0:,0:])".
```

```
        /// </summary>
        /// <param name="array1">Describe parameter.</param>
        /// <param name="array">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int gg(short[] array1, int[,] array) { return 0; }

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Addition(N.X,N.X)".
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <param name="xx">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static X operator +(X x, X xx) { return x; }

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.prop".
        /// </summary>
        public int prop { get { return 1; } set { } }

        /// <summary>
        /// Enter description for event.
        /// ID string generated is "E:N.X.d".
        /// </summary>
        public event D d;

        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.Item(System.String)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <returns></returns>
        public int this[string s] { get { return 1; } }

        /// <summary>
        /// Enter description for class Nested.
        /// ID string generated is "T:N.X.Nested".
        /// </summary>
        public class Nested { }

        /// <summary>
        /// Enter description for delegate.
        /// ID string generated is "T:N.X.D".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public delegate void D(int i);

        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Explicit(N.X)~System.Int32".
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static explicit operator int(X x) { return 1; }
    }
}
```

Vea también

- Guía de programación de C#
- -doc (opciones del compilador de C#)
- Comentarios de documentación XML

Delimitadores de etiquetas de documentación (guía de programación de C#)

29/06/2020 • 5 minutes to read • Edit Online

El uso de comentarios de documentación XML requiere delimitadores, que le indican al compilador dónde empieza y dónde acaba un comentario de documentación. Puede usar los siguientes tipos de delimitadores con las etiquetas de documentación XML:

- `///`

  Delimitador de una sola línea. Este es el formato que se muestra en los ejemplos de documentación y que usan las plantillas de proyecto de C#. Si hay un carácter de espacio en blanco después del delimitador, ese carácter no se incluye en la salida XML.

  > **NOTE**
  > El entorno de desarrollo integrado (IDE) de Visual Studio inserta automáticamente las etiquetas `<summary>` y `</summary>` y mueve el cursor dentro de estas etiquetas después de escribir el delimitador `///` en el editor de código. Puede activar o desactivar esta característica en el cuadro de diálogo Opciones.

- `/** */`

  Delimitadores de múltiples líneas.

  Debe seguir ciertas reglas de formato cuando use los delimitadores `/** */`:

  - En la línea que contiene el delimitador `/**`, si el resto de la línea es espacio en blanco, la línea no se procesa en busca de comentarios. Si el primer carácter después del delimitador `/**` es un espacio en blanco, dicho carácter de espacio en blanco se omite y se procesa el resto de la línea. En caso contrario, todo el texto de la línea situado después del delimitador `/**` se procesa como parte del comentario.

  - En la línea que contiene el delimitador `*/`, si solo hay espacio en blanco hasta el delimitador `*/`, esa línea se omite. En caso contrario, el texto de la línea situado antes del delimitador `*/` se procesa como parte del comentario.

  - Para las líneas que aparecen después de la que comienza con el delimitador `/**`, el compilador busca un patrón común al principio de cada línea. El patrón puede consistir en un espacio en blanco opcional y un asterisco (`*`), seguido de otro espacio en blanco opcional. Si el compilador encuentra un patrón común al principio de cada línea que no empieza por el delimitador `/**` o el delimitador `*/`, omite ese patrón para cada línea.

  En los siguientes ejemplos se muestran estas reglas.

  - La única parte del comentario siguiente que se procesará es la línea que comienza con `<summary>`. Los tres formatos de etiquetas producen los mismos comentarios.

```
/** <summary>text</summary> */

/**
<summary>text</summary>
*/

/**
* <summary>text</summary>
*/
```

- El compilador identifica un patrón común de " * " al principio de la segunda y la tercera línea. El patrón no se incluye en la salida.

```
/**
* <summary>
* text </summary>*/
```

- El compilador no encuentra ningún patrón común en el comentario siguiente porque el segundo carácter de la tercera línea no es un asterisco. Por lo tanto, todo el texto de la segunda y la tercera línea se procesa como parte del comentario.

```
/**
* <summary>
   text </summary>
*/
```

- El compilador no encuentra ningún patrón en el comentario siguiente por dos razones. En primer lugar, el número de espacios antes del asterisco no es coherente. En segundo lugar, la quinta línea comienza con una tabulación, por lo tanto los espacios no coinciden. Como resultado, todo el texto de las líneas de la dos a la cinco se procesa como parte del comentario.

```
/**
  * <summary>
  * text
*  text2
    *  </summary>
*/
```

Vea también

- Guía de programación de C#
- Comentarios de documentación XML
- -doc (opciones del compilador de C#)

Procedimiento para usar las características de la documentación XML

29/06/2020 • 5 minutes to read • Edit Online

En el ejemplo siguiente se proporciona una introducción básica de un tipo que se ha documentado.

Ejemplo

```
// If compiling from the command line, compile with: -doc:YourFileName.xml

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.
/// </remarks>
public class TestClass : TestInterface
{
    /// <summary>
    /// Store for the Name property.
    /// </summary>
    private string _name = null;

    /// <summary>
    /// The class constructor.
    /// </summary>
    public TestClass()
    {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (_name == null)
            {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>
    /// Description for SomeMethod.
    /// </summary>
    /// <param name="s"> Parameter description for s goes here.</param>
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a type or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s)
    {
```

```
    }

    /// <summary>
    /// Some other method.
    /// </summary>
    /// <returns>
    /// Return values are described through the returns tag.
    /// </returns>
    /// <seealso cref="SomeMethod(string)">
    /// Notice the use of the cref attribute to reference a specific method.
    /// </seealso>
    public int SomeOtherMethod()
    {
        return 0;
    }

    public int InterfaceMethod(int n)
    {
        return n * n;
    }

    /// <summary>
    /// The entry point for the application.
    /// </summary>
    /// <param name="args"> A list of command line arguments.</param>
    static int Main(System.String[] args)
    {
        // TODO: Add code to start application here.
        return 0;
    }
}

/// <summary>
/// Documentation that describes the interface goes here.
/// </summary>
/// <remarks>
/// Details about the interface go here.
/// </remarks>
interface TestInterface
{
    /// <summary>
    /// Documentation that describes the method goes here.
    /// </summary>
    /// <param name="n">
    /// Parameter n requires an integer argument.
    /// </param>
    /// <returns>
    /// The method returns an integer.
    /// </returns>
    int InterfaceMethod(int n);
}
```

En el ejemplo se genera un archivo *.xml* con el contenido siguiente.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xmlsample</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            <summary>
            Class level summary documentation goes here.
            </summary>
            <remarks>
            Longer comments can be associated with a type or member through
            the remarks tag.
```

```
            </remarks>
        </member>
        <member name="F:TestClass._name">
            <summary>
            Store for the Name property.
            </summary>
        </member>
        <member name="M:TestClass.#ctor">
            <summary>
            The class constructor.
            </summary>
        </member>
        <member name="P:TestClass.Name">
            <summary>
            Name property.
            </summary>
            <value>
            A value tag is used to describe the property value.
            </value>
        </member>
        <member name="M:TestClass.SomeMethod(System.String)">
            <summary>
            Description for SomeMethod.
            </summary>
            <param name="s"> Parameter description for s goes here.</param>
            <seealso cref="T:System.String">
            You can use the cref attribute on any tag to reference a type or member
            and the compiler will check that the reference exists.
            </seealso>
        </member>
        <member name="M:TestClass.SomeOtherMethod">
            <summary>
            Some other method.
            </summary>
            <returns>
            Return values are described through the returns tag.
            </returns>
            <seealso cref="M:TestClass.SomeMethod(System.String)">
            Notice the use of the cref attribute to reference a specific method.
            </seealso>
        </member>
        <member name="M:TestClass.Main(System.String[])">
            <summary>
            The entry point for the application.
            </summary>
            <param name="args"> A list of command line arguments.</param>
        </member>
        <member name="T:TestInterface">
            <summary>
            Documentation that describes the interface goes here.
            </summary>
            <remarks>
            Details about the interface go here.
            </remarks>
        </member>
        <member name="M:TestInterface.InterfaceMethod(System.Int32)">
            <summary>
            Documentation that describes the method goes here.
            </summary>
            <param name="n">
            Parameter n requires an integer argument.
            </param>
            <returns>
            The method returns an integer.
            </returns>
        </member>
    </members>
</doc>
```

Compilación del código

Para compilar el ejemplo, escriba el siguiente comando:

```
csc XMLsample.cs /doc:XMLsample.xml
```

Este comando crea el archivo XML *XMLsample.xml*, que se puede ver en el explorador o mediante el comando `TYPE`.

Programación sólida

La documentación XML empieza con `///`. Cuando se crea un proyecto, el asistente agrega automáticamente unas líneas de inicio `///`. El procesamiento de estos comentarios tiene algunas restricciones:

- La documentación debe ser XML con formato correcto. Si el XML no tiene el formato correcto, se generará una advertencia y el archivo de documentación incluirá un comentario en el que se indica que se detectó un error.
- Los desarrolladores pueden crear su propio conjunto de etiquetas, Hay un conjunto recomendado de etiquetas. Algunas de las etiquetas recomendadas tienen significados especiales:
  - La etiqueta `<param>` se usa para describir parámetros. Si se usa, el compilador comprueba que el parámetro existe y que todos los parámetros se describen en la documentación. Si se produce un error en la comprobación, el compilador emite una advertencia.
  - El atributo `cref` se puede asociar a cualquier etiqueta para hacer referencia a un elemento de código. El compilador comprueba si existe este elemento de código. Si se produce un error en la comprobación, el compilador emite una advertencia. El compilador respeta todas las instrucciones `using` cuando busca un tipo descrito en el atributo `cref`.
  - IntelliSense usa la etiqueta `<summary>` en Visual Studio para mostrar información adicional sobre un tipo o miembro.

    > **NOTE**
    > El archivo XML no proporciona información completa sobre los tipos y los miembros (por ejemplo, no contiene información de tipos). Para obtener información completa sobre un tipo o miembro, use el archivo de documentación con reflexión en el tipo o miembro reales.

Vea también

- Guía de programación de C#
- -doc (opciones del compilador de C#)
- Comentarios de documentación XML
- Procesador de documentación de DocFX
- Procesador de documentación de Sandcastle

<c> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<c>text</c>
```

Parámetros

- `text`

  El texto que le gustaría indicar como código.

Comentarios

La etiqueta `<c>` le proporciona una manera de indicar que el texto dentro de una descripción debe marcarse como código. Use <code> para indicar varias líneas como código.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary><c>DoWork</c> is a method in the <c>TestClass</c> class.
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<code> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<code>content</code>
```

Parámetros

- `content`

  El texto que quiere marcar como código.

Comentarios

La etiqueta `<code>` se usa para indicar varias líneas de código. Use <c> para indicar que el texto de línea única dentro de una descripción debe marcarse como código.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

Vea el artículo <example> para obtener un ejemplo de cómo usar la etiqueta `<code>`.

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

Atributo cref (guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

El atributo `cref` en una etiqueta de documentación XML significa "referencia de código". Especifica que el texto interno de la etiqueta es un elemento de código, como un tipo, un método o una propiedad. En herramientas de documentación como DocFX y Sandcastle, use los atributos `cref` para generar hipervínculos a la página donde se documenta el tipo o miembro de manera automática.

Ejemplo

En el ejemplo siguiente se muestran los atributos `cref` que se usan en las etiquetas `<see>`.

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>
```

```
        public static T GetGenericValue<T>(T para)
        {
            return para;
        }
    }

    /// <summary>
    /// GenericClass.
    /// </summary>
    /// <remarks>
    /// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
    /// </remarks>
    class GenericClass<T>
    {
        // Fields and members.
    }

    class Program
    {
        static int Main()
        {
            return TestClass.GetZero();
        }
    }
}
```

Cuando se compila, el programa genera el archivo XML siguiente. Observe que, por ejemplo, el compilador ha transformado el atributo `cref` del método `GetZero` en `"M:TestNamespace.TestClass.GetZero"`. El prefijo "M:" significa "método", una convención reconocida por las herramientas de documentación tales como DocFX y Sandcastle. Para obtener una lista completa de los prefijos, vea Procesar el archivo XML.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
            TestClass contains several cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor">
            <summary>
            This sample shows how to specify the <see cref="T:TestNamespace.TestClass"/> constructor as a cref
attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor(System.Int32)">
            <summary>
            This sample shows how to specify the <see cref="M:TestNamespace.TestClass.#ctor(System.Int32)"/>
constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
            The GetZero method.
            </summary>
            <example>
            This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/> method.
            <code>
            class TestClass
            {
                static int Main()
                {
                    return GetZero();
                }
            }
            </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue``1(``0)">
            <summary>
            The GetGenericValue method.
            </summary>
            <remarks>
            This sample shows how to specify the <see
cref="M:TestNamespace.TestClass.GetGenericValue``1(``0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
            GenericClass.
            </summary>
            <remarks>
            This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref
attribute.
            </remarks>
        </member>
    </members>
</doc>
```

Vea también

- Comentarios de documentación XML
- Etiquetas recomendadas para los comentarios de documentación

<example> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<example>description</example>
```

Parámetros

- `description`

  Una descripción del ejemplo de código.

Comentarios

La etiqueta `<example>` le permite especificar un ejemplo de cómo usar un método u otro miembro de biblioteca. Esto normalmente implica el uso de la etiqueta <code>.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
```

```
        /// }
        /// </code>
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>

        public static T GetGenericValue<T>(T para)
        {
            return para;
        }
    }

    /// <summary>
    /// GenericClass.
    /// </summary>
    /// <remarks>
    /// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
    /// </remarks>
    class GenericClass<T>
    {
        // Fields and members.
    }

    class Program
    {
        static int Main()
        {
            return TestClass.GetZero();
        }
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<exception> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<exception cref="member">description</exception>
```

Parámetros

- cref = "`member`"

  Una referencia a una excepción que está disponible desde el entorno de compilación actual. El compilador comprueba si la excepción dada existe y traduce `member` al nombre de elemento canónico en la salida XML. `member` debe aparecer entre comillas dobles (" ").

  Para más información sobre cómo dar formato a `member` para hacer referencia a un tipo genérico, consulte Procesar el archivo XML.

- `description`

  Descripción de la excepción.

Comentarios

La etiqueta `<exception>` le permite especificar qué excepciones se pueden producir. Esta etiqueta se puede aplicar a definiciones de métodos, propiedades, eventos e indizadores.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Para más información sobre el control de excepciones, vea Excepciones y control de excepciones.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// Comment for class
public class EClass : System.Exception
{
    // class definition...
}

/// Comment for class
class TestClass
{
    /// <exception cref="System.Exception">Thrown when...</exception>
    public void DoSomething()
    {
        try
        {
        }
        catch (EClass)
        {
        }
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<include> (guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

Sintaxis

```
<include file='filename' path='tagpath[@name="id"]' />
```

Parámetros

- `filename`

  El nombre del archivo XML que contiene la documentación. El nombre de archivo se puede calificar con una ruta de acceso relativa al archivo de código fuente. Incluya `filename` entre comillas simples (' ').

- `tagpath`

  La ruta de acceso de las etiquetas de `filename` que conduce a la etiqueta `name`. Incluya la ruta de acceso entre comillas simples (' ').

- `name`

  El especificador de nombre en la etiqueta que precede a los comentarios; `name` tendrá un `id`.

- `id`

El identificador de la etiqueta que precede a los comentarios. Ponga el identificador entre comillas dobles (" ").

Comentarios

La etiqueta `<include>` le permite hacer referencia a comentarios colocados en otro archivo que describen los tipos y miembros en el código fuente. Esto es una alternativa a colocar los comentarios de documentación directamente en el archivo de código fuente. Al colocar la documentación en un archivo independiente, puede aplicar el control de código fuente a la documentación de forma independiente desde el código fuente. Una persona puede tener el archivo de código fuente extraído del repositorio y otra persona puede tener el archivo de documentación extraído del repositorio.

La etiqueta `<include>` usa la sintaxis de XPath de XML. Consulte la documentación de XPath para ver formas de personalizar el uso de `<include>`.

Ejemplo

Este es un ejemplo de múltiples archivos. El siguiente es el primer archivo, que usa `<include>`.

```
// compile with: -doc:DocFileName.xml

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

El segundo archivo, *xml_include_tag.doc*, contiene los comentarios de documentación siguientes.

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

Salida del programa

El siguiente resultado se genera al compilar las clases Test y Test2 con la siguiente línea de comandos: `-doc:DocFileName.xml.` En Visual Studio, especifica la opción de comentarios de documentación XML en el panel Compilar del Diseñador de proyectos. Cuando el compilador de C# ve la etiqueta `<include>`, busca los comentarios de documentación en *xml_include_tag.doc* en lugar del archivo de código fuente actual. Después, el compilador genera *DocFileName.xml*, y este es el archivo que usan las herramientas de documentación como DocFX y Sandcastle para generar la documentación final.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xml_include_tag</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
</summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
</summary>
        </member>
    </members>
</doc>
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<inheritdoc> (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<inheritdoc/>
```

InheritDoc

Herede comentarios XML de clases base, interfaces y métodos similares. Esto elimina la acción de copiar y pegar no deseada de comentarios XML duplicados y mantiene los comentarios XML sincronizados automáticamente.

Comentarios

Agregue sus comentarios XML en las clases base o las interfaces y deje que InheritDoc copie los comentarios en las clases en implementación.

Agregue sus comentarios XML a los métodos sincrónicos y deje que InheritDoc copie los comentarios en las versiones asincrónicas de los mismos métodos.

Si quiere copiar los comentarios de un miembro en concreto, puede utilizar el atributo `cref` para especificar el miembro.

Ejemplos

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
public class MainClass
{
}

///<inheritdoc/>
public class TestClass: MainClass
{
}
```

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this interface.
/// </summary>
public interface ITestInterface
{
}

///<inheritdoc cref="ITestInterface"/>
public class TestClass : ITestInterface
{
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<list> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<list type="bullet|number|table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>term</term>
        <description>description</description>
    </item>
</list>
```

Parámetros

- `term`

  Término que se define en `description`.

- `description`

  Elemento de una lista numerada o con viñetas, o definición de un `term`.

Comentarios

El bloque `<listheader>` se usa para definir la fila de encabezado de una tabla o de una lista de definiciones. Cuando se define una tabla, solo es necesario suministrar una entrada para un término en el encabezado.

Cada elemento de la lista se especifica con un bloque `<item>`. Cuando se crea una lista de definiciones, se deberán especificar tanto `term` como `description`. En cambio, para una tabla, lista con viñetas o lista numerada, solo es necesario suministrar una entrada para `description`.

Una lista o una tabla pueden tener tantos bloques `<item>` como sean necesarios.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<para> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<para>content</para>
```

Parámetros

- `content`

  El texto del párrafo.

Comentarios

La etiqueta `<para>` se usa dentro de otra etiqueta, como <summary>, <remarks> o <returns>, y permite dar una estructura al texto.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

Vea <summary> para obtener un ejemplo en el que se usa `<para>`.

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<param> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<param name="name">description</param>
```

Parámetros

- `name`

  Nombre de un parámetro de método. Ponga el nombre entre comillas dobles (" ").

- `description`

  Descripción del parámetro.

Comentarios

La etiqueta `<param>` debe usarse en el comentario de una declaración de método para describir uno de los parámetros del método. Para documentar varios parámetros, use varias etiquetas `<param>`.

El texto de la etiqueta `<param>` se muestra en IntelliSense, el examinador de objetos y el informe web de comentario de código.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    // Single parameter.
    /// <param name="Int1">Used to indicate status.</param>
    public static void DoWork(int Int1)
    {
    }

    // Multiple parameters.
    /// <param name="Int1">Used to indicate status.</param>
    /// <param name="Float1">Used to specify context.</param>
    public static void DoWork(int Int1, float Float1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<paramref> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<paramref name="name"/>
```

Parámetros

- `name`

  Nombre del parámetro al que se hace referencia. Ponga el nombre entre comillas dobles (" ").

Comentarios

La etiqueta `<paramref>` ofrece una manera de indicar que una palabra en los comentarios del código (por ejemplo, en un bloque `<summary>` o `<remarks>`) hace referencia a un parámetro. El archivo XML se puede procesar para dar formato a esta palabra de alguna manera distinta, por ejemplo, con una fuente en negrita o cursiva.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// The <paramref name="int1"/> parameter takes a number.
    /// </summary>
    public static void DoWork(int int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<permission> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<permission cref="member">description</permission>
```

Parámetros

- cref = "`member`"

  Referencia a un miembro o campo al cual se puede llamar desde el entorno de compilación actual. El compilador comprueba si el elemento de código dado existe y traduce `member` al nombre de elemento canónico en la salida XML. *member* debe aparecer entre comillas dobles (" ").

  Para obtener información sobre cómo crear una referencia cref a un tipo genérico, vea Atributo cref.

- `description`

  Descripción del acceso al miembro.

Comentarios

La etiqueta `<permission>` le permite documentar el acceso de un miembro. La clase PermissionSet le permite especificar el acceso a un miembro.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

class TestClass
{
    /// <permission cref="System.Security.PermissionSet">Everyone can access this method.</permission>
    public static void Test()
    {
    }

    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<remarks> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<remarks>description</remarks>
```

Parámetros

- `Description`

  Descripción del miembro.

Comentarios

La etiqueta `<remarks>` se usa para agregar información sobre un tipo y complementa la información especificada con <summary>. Esta información se muestra en la ventana Examinador de objetos.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class TestClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<returns> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<returns>description</returns>
```

Parámetros

- `description`

  Descripción del valor devuelto.

Comentarios

La etiqueta `<returns>` debe usarse en el comentario de una declaración de método para describir el valor devuelto.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <returns>Returns zero.</returns>
    public static int GetZero()
    {
        return 0;
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<see> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<see cref="member"/>
```

Parámetros

- cref = "`member`"

  Referencia a un miembro o campo al cual se puede llamar desde el entorno de compilación actual. El compilador comprueba si el elemento de código dado existe y pasa `member` al nombre de elemento en el resultado XML. Agregue *member* entre comillas dobles (" ").

Comentarios

La etiqueta `<see>` permite especificar un vínculo desde el texto. Use <seealso> para indicar que el texto debe colocarse en una sección Vea también. Use el atributo cref para crear hipervínculos internos a las páginas de documentación de los elementos de código.

Compile con -doc para procesar los comentarios de documentación de un archivo.

En el ejemplo siguiente, se muestra una etiqueta `<see>` dentro de una sección de resumen.

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<seealso> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<seealso cref="member"/>
```

Parámetros

- cref = "`member`"

  Referencia a un miembro o campo al cual se puede llamar desde el entorno de compilación actual. El compilador comprueba si el elemento de código dado existe y pasa `member` al nombre de elemento en el resultado XML. `member` debe aparecer entre comillas dobles (" ").

  Para obtener información sobre cómo crear una referencia cref a un tipo genérico, vea Atributo cref.

Comentarios

La etiqueta `<seealso>` le permite especificar el texto que quiere que aparezca en una sección Vea también. Use <see> para especificar un vínculo desde dentro del texto.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

Vea <summary> para obtener un ejemplo en el que se usa <seealso>.

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<summary> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<summary>description</summary>
```

Parámetros

- `description`

  Resumen del objeto.

Comentarios

La etiqueta `<summary>` debe usarse para describir un tipo o un miembro de tipo. Use <remarks> para agregar información adicional a una descripción de tipo. Use el atributo cref para permitir que herramientas de documentación como DocFX y Sandcastle creen hipervínculos internos a las páginas de documentación de los elementos de código.

El texto de la etiqueta `<summary>` es la única fuente de información sobre el tipo en IntelliSense y también se muestra en la ventana Examinador de objetos.

Compile con -doc para procesar los comentarios de documentación de un archivo. Para crear la documentación final basada en el archivo generado por el compilador, puede crear una herramienta personalizada o usar una herramienta como DocFX o Sandcastle.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

En el ejemplo anterior se genera el siguiente archivo XML.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>YourNamespace</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            text for class TestClass
        </member>
        <member name="M:TestClass.DoWork(System.Int32)">
            <summary>DoWork is a method in the TestClass class.
            <para>Here's how you could make a second paragraph in a description. <see
cref="M:System.Console.WriteLine(System.String)"/> for information about output statements.</para>
            </summary>
            <seealso cref="M:TestClass.Main"/>
        </member>
        <member name="M:TestClass.Main">
            text for Main
        </member>
    </members>
</doc>
```

Ejemplo

En el ejemplo siguiente se muestra cómo hacer una referencia `cref` a un tipo genérico.

```
// compile with: -doc:DocFileName.xml

// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <summary cref="C{T}">
/// </summary>
class A { }

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
// <summary cref="C &lt; T &gt;">

// the following cref shows how to hard-code the reference
/// <summary cref="T:C`1">
/// </summary>
class B { }

/// <summary cref="A">
/// </summary>
/// <typeparam name="T"></typeparam>
class C<T> { }

class Program
{
    static void Main() { }
}
```

En el ejemplo anterior se genera el siguiente archivo XML.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:A">
            <summary cref="T:C`1">
            </summary>
        </member>
        <member name="T:B">
            <summary cref="T:C`1">
            </summary>
        </member>
        <member name="T:C`1">
            <summary cref="T:A">
            </summary>
            <typeparam name="T"></typeparam>
        </member>
    </members>
</doc>
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<typeparam> (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<typeparam name="name">description</typeparam>
```

Parámetros

- `name`

  Nombre del parámetro de tipo. Ponga el nombre entre comillas dobles (" ").

- `description`

  Una descripción del parámetro de tipo.

Comentarios

La etiqueta `<typeparam>` debe usarse en el comentario de una declaración de método o tipo genérico para describir un parámetro de tipo. Agregue una etiqueta para cada parámetro de tipo del tipo o método genérico.

Para más información, vea Genéricos.

El texto de la etiqueta `<typeparam>` se mostrará en IntelliSense, el informe web de comentario de código de la Ventana Examinador de objetos.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

Vea también

- Referencia de C#
- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<typeparamref> (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<typeparamref name="name"/>
```

Parámetros

- `name`

  Nombre del parámetro de tipo. Ponga el nombre entre comillas dobles (" ").

Comentarios

Para obtener más información sobre los parámetros de tipo en métodos y tipos genéricos, vea Genéricos.

Use esta etiqueta para permitir que los consumidores del archivo de documentación den formato a la palabra de alguna manera distinta, por ejemplo en cursiva.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

<value> (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Sintaxis

```
<value>property-description</value>
```

Parámetros

- `property-description`

  Una descripción de la propiedad.

Comentarios

La etiqueta `<value>` le permite describir el valor que representa una propiedad. Cuando agrega una propiedad mediante un asistente de código en el entorno de desarrollo .NET de Visual Studio, agregará una etiqueta <summary> para la nueva propiedad. Después, debe agregar manualmente una etiqueta `<value>` para describir el valor que representa esa propiedad.

Compile con -doc para procesar los comentarios de documentación de un archivo.

Ejemplo

```
// compile with: -doc:DocFileName.xml

/// text for class Employee
public class Employee
{
    private string _name;

    /// <summary>The Name property represents the employee's name.</summary>
    /// <value>The Name property gets/sets the value of the string field, _name.</value>

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}

/// text for class MainClass
public class MainClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

Vea también

- Guía de programación de C#
- Etiquetas recomendadas para los comentarios de documentación

Excepciones y control de excepciones (Guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

Las características de control de excepciones del lenguaje C# le ayudan a afrontar cualquier situación inesperada o excepcional que se produce cuando se ejecuta un programa. El control de excepciones usa las palabras clave `try`, `catch` y `finally` para intentar realizar acciones que pueden no completarse correctamente, para controlar errores cuando decide que es razonable hacerlo y para limpiar recursos más adelante. Las excepciones las puede generar Common Language Runtime (CLR), .NET, bibliotecas de terceros o el código de aplicación. Las excepciones se crean mediante el uso de la palabra clave `throw`.

En muchos casos, una excepción la puede no producir un método al que el código ha llamado directamente, sino otro método más bajo en la pila de llamadas. Cuando esto sucede, CLR desenredar la pila, busca un método con un bloque `catch` para el tipo de excepción específico y ejecuta el primer bloque `catch` que encuentra. Si no encuentra ningún bloque `catch` adecuado en cualquier parte de la pila de llamadas, finalizará el proceso y mostrará un mensaje al usuario.

En este ejemplo, un método prueba a hacer la división entre cero y detecta el error. Sin el control de excepciones, este programa finalizaría con un error **DivideByZeroException no controlada**.

```
class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Información general sobre excepciones

Las excepciones tienen las siguientes propiedades:

- Las excepciones son tipos que derivan en última instancia de `System.Exception`.
- Use un bloque `try` alrededor de las instrucciones que pueden producir excepciones.
- Una vez que se produce una excepción en el bloque `try`, el flujo de control salta al primer controlador de

excepciones asociado que está presente en cualquier parte de la pila de llamadas. En C#, la palabra clave `catch` se utiliza para definir un controlador de excepciones.

- Si no hay ningún controlador de excepciones para una excepción determinada, el programa deja de ejecutarse con un mensaje de error.
- No detecte una excepción a menos que pueda controlarla y dejar la aplicación en un estado conocido. Si se detecta `System.Exception`, reinícielo con la palabra clave `throw` al final del bloque `catch`.
- Si un bloque `catch` define una variable de excepción, puede utilizarla para obtener más información sobre el tipo de excepción que se ha producido.
- Las excepciones puede generarlas explícitamente un programa con la palabra clave `throw`.
- Los objetos de excepción contienen información detallada sobre el error, como el estado de la pila de llamadas y una descripción de texto del error.
- El código de un bloque `finally` se ejecuta incluso si se produce una excepción. Use un bloque `finally` para liberar recursos, por ejemplo, para cerrar las secuencias o los archivos que se abrieron en el bloque `try`.
- Las excepciones administradas de .NET se implementan en el mecanismo de control de excepciones estructurado de Win32. Para más información, vea Control de excepciones estructurado (C/C++) y A Crash Course on the Depths of Win32 Structured Exception Handling (Curso intensivo sobre los aspectos específicos del control de excepciones estructurado de Win32).

Secciones relacionadas

Para obtener más información sobre las excepciones y el control de excepciones, vea los artículos siguientes:

- Utilizar excepciones
- Control de excepciones
- Crear y producir excepciones
- Excepciones generadas por el compilador
- Procedimiento para controlar una excepción mediante Try y Catch (Guía de programación de C#)
- Procedimiento para ejecutar código de limpieza mediante finally
- Procedimiento para detectar excepciones no compatibles con CLS

Especificación del lenguaje C#

Para obtener más información, consulte la sección Excepciones de Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- SystemException
- Guía de programación de C#
- Palabras clave de C#
- throw
- try-catch
- try-finally
- try-catch-finally
- Excepciones

Uso de excepciones (guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

En C#, los errores del programa en tiempo de ejecución se propagan a través del programa mediante un mecanismo denominado excepciones. Las excepciones las inicia el código que encuentra un error y las detecta el código que puede corregir dicho error. El entorno de ejecución .NET o el código de un programa pueden producir excepciones. Una vez iniciada, una excepción se propaga hasta la pila de llamadas hasta que encuentra una instrucción `catch` para la excepción. Las excepciones no detectadas se controlan mediante un controlador de excepciones que ofrece el sistema y muestra un cuadro de diálogo.

Las excepciones están representadas por clases derivadas de Exception. Esta clase identifica el tipo de excepción y contiene propiedades que tienen los detalles sobre la excepción. Iniciar una excepción implica crear una instancia de una clase derivada de excepción, configurar opcionalmente las propiedades de la excepción y luego producir el objeto con la palabra clave `throw`. Por ejemplo:

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    CustomException ex =
        new CustomException("Custom exception in TestThrow()");

    throw ex;
}
```

Cuando se inicia una excepción, el entorno runtime comprueba la instrucción actual para ver si se encuentra dentro de un bloque `try`. Si es así, se comprueban los bloques `catch` asociados al bloque `try` para ver si pueden detectar la excepción. Los bloques `Catch` suelen especificar tipos de excepción; si el tipo del bloque `catch` es el mismo de la excepción, o una clase base de la excepción, el bloque `catch` puede controlar el método. Por ejemplo:

```
static void TestCatch()
{
    try
    {
        TestThrow();
    }
    catch (CustomException ex)
    {
        System.Console.WriteLine(ex.ToString());
    }
}
```

Si la instrucción que inicia una excepción no está en un bloque `try` o si el bloque `try` que la encierra no tiene un bloque `catch` coincidente, el entorno runtime busca una instrucción `try` y bloques `catch` en el método de llamada. El entorno runtime sigue hasta la pila de llamadas para buscar un bloque `catch` compatible. Después de encontrar el bloque `catch` y ejecutarlo, el control pasa a la siguiente instrucción después de dicho bloque `catch`.

Una instrucción `try` puede contener más de un bloque `catch`. Se ejecuta la primera instrucción `catch` que

pueda controlar la excepción; las instrucciones `catch` siguientes se omiten, aunque sean compatibles. Por consiguiente, los bloques catch deben ordenarse siempre de más específico (o más derivado) a menos específico. Por ejemplo:

```
using System;
using System.IO;

public class ExceptionExample
{
    static void Main()
    {
        try
        {
            using (var sw = new StreamWriter(@"C:\test\test.txt"))
            {
                sw.WriteLine("Hello");
            }
        }
        // Put the more specific exceptions first.
        catch (DirectoryNotFoundException ex)
        {
            Console.WriteLine(ex);
        }
        catch (FileNotFoundException ex)
        {
            Console.WriteLine(ex);
        }
        // Put the least specific exception last.
        catch (IOException ex)
        {
            Console.WriteLine(ex);
        }

        Console.WriteLine("Done");
    }
}
```

Para que el bloque `catch` se ejecute, el entorno runtime busca bloques `finally`. Los bloques `Finally` permiten al programador limpiar cualquier estado ambiguo que pudiera haber quedado tras la anulación de un bloque `try` o liberar los recursos externos (como identificadores de gráficos, conexiones de base de datos o flujos de archivos) sin tener que esperar a que el recolector de elementos no utilizados del entorno runtime finalice los objetos. Por ejemplo:

```
static void TestFinally()
{
    System.IO.FileStream file = null;
    //Change the path to something that works on your machine.
    System.IO.FileInfo fileInfo = new System.IO.FileInfo(@"C:\file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise IOException is thrown.
        if (file != null)
        {
            file.Close();
        }
    }

    try
    {
        file = fileInfo.OpenWrite();
        System.Console.WriteLine("OpenWrite() succeeded");
    }
    catch (System.IO.IOException)
    {
        System.Console.WriteLine("OpenWrite() failed");
    }
}
```

Si `WriteByte()` ha iniciado una excepción, el código del segundo bloque `try` que intente reabrir el archivo generaría un error si no se llama a `file.Close()` y el archivo permanecería bloqueado. Como los bloques `finally` se ejecutan aunque se inicie una excepción, el bloque `finally` del ejemplo anterior permite que el archivo se cierre correctamente y ayuda a evitar un error.

Si no se encuentra ningún bloque `catch` compatible en la pila de llamadas después de iniciar una excepción, sucede una de estas tres acciones:

- Si la excepción se encuentra en un finalizador, este se anula y, si procede, se llama al finalizador base.
- Si la pila de llamadas contiene un constructor estático o un inicializador de campo estático, se inicia una excepción TypeInitializationException, y la excepción original se asigna a la propiedad InnerException de la nueva excepción.
- Si se llega al comienzo del subproceso, este finaliza.

Vea también

- Guía de programación de C#
- Excepciones y control de excepciones

Control de excepciones (Guía de programación de C#)

18/03/2020 • 6 minutes to read • Edit Online

Los programadores de C# usan un bloque try para separar el código que podría verse afectado por una excepción. Los bloques catch asociados se usan para controlar las excepciones resultantes. Los bloques finally contienen código que se ejecuta independientemente de si se produce una excepción en el bloque `try`, como la liberación de recursos asignados en el bloque `try`. Los bloques `try` requieren uno o varios bloques `catch` asociados, un bloque `finally` o ambos.

En los ejemplos siguientes se muestra una instrucción `try-catch`, una instrucción `try-finally` y una instrucción `try-catch-finally`.

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

Un bloque `try` sin un bloque `catch` o `finally` produce un error del compilador.

Bloques catch

Los bloques `catch` pueden especificar el tipo de excepción que quiere detectar. La especificación de tipo se

denomina *filtro de excepciones*. El tipo de excepción se debe derivar de Exception. En general, no especifique Exception como el filtro de excepciones a menos que sepa cómo controlar todas las excepciones que puedan producirse en el bloque `try` o que haya incluido una instrucción throw al final del bloque `catch`.

Se pueden encadenar juntos varios bloques `catch` con distintos filtros de excepciones. Los bloques `catch` se evalúan de arriba abajo en el código, pero solo se ejecuta un bloque `catch` para cada excepción que se produce. Se ejecuta el primer bloque `catch` que especifica el tipo exacto o una clase base de la excepción producida. Si no hay ningún bloque `catch` que especifique un filtro de excepciones coincidente, se selecciona un bloque `catch` sin filtros, si hay alguno en la instrucción. Es importante colocar primero los bloques `catch` con los tipos de excepción más específicos (es decir, los más derivados).

Debe detectar excepciones cuando se cumplan las siguientes condiciones:

- Comprende bien el motivo por el que podría producirse la excepción y puede implementar una recuperación específica, por ejemplo, pedir al usuario que escriba un nuevo nombre de archivo cuando detecte un objeto FileNotFoundException.
- Puede crear y producir una nueva excepción más específica.

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch(System.IndexOutOfRangeException e)
    {
        throw new System.ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

- Quiere controlar parcialmente una excepción antes de pasarla para su control adicional. En el ejemplo siguiente, se usa un bloque `catch` para agregar una entrada a un registro de errores antes de volver a producir la excepción.

```
try
{
    // Try to access a resource.
}
catch (System.UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

Bloques Finally

Los bloques `finally` permiten limpiar las acciones que se realizan en un bloque `try`. Si está presente, el bloque `finally` se ejecuta en último lugar, después del bloque `try` y de cualquier bloque `catch` coincidente. Los bloques `finally` siempre se ejecutan, independientemente de si se produce una excepción o si se encuentra un bloque `catch` que coincida con el tipo de excepción.

Los bloques `finally` pueden usarse para liberar recursos como secuencias de archivo, conexiones de base de datos y controladores de gráficos sin necesidad de esperar a que el recolector de elementos no utilizados en tiempo de ejecución finalice los objetos. Consulte using (Instrucción) para obtener más información.

En el ejemplo siguiente, el bloque `finally` se usa para cerrar un archivo que se abre en el bloque `try`. Observe que se comprueba el estado del identificador de archivos antes de cerrar el archivo. Si el bloque `try` no puede abrir el archivo, el identificador de archivos sigue teniendo el valor `null` y el bloque `finally` no intenta cerrarlo. Como alternativa, si el archivo se abre correctamente en el bloque `try`, el bloque `finally` cierra el archivo abierto.

```
System.IO.FileStream file = null;
System.IO.FileInfo fileinfo = new System.IO.FileInfo("C:\\file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    if (file != null)
    {
        file.Close();
    }
}
```

Especificación del lenguaje C#

Para obtener más información, vea las secciones Excepciones y La instrucción try de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Excepciones y control de excepciones
- try-catch
- try-finally
- try-catch-finally
- using (instrucción)

Crear y producir excepciones (Guía de programación de C#)

18/03/2020 • 7 minutes to read • Edit Online

Las excepciones se usan para indicar que se ha producido un error mientras se ejecutaba el programa. Se crean los objetos de excepción que describen un error y, luego, se *producen* con la palabra clave throw. Después, el tiempo de ejecución busca el controlador de excepciones más compatible.

Los programadores deberían producir excepciones cuando una o varias de las siguientes condiciones sean verdaderas:

- El método no puede finalizar su funcionalidad definida.

  Por ejemplo, si un parámetro de un método tiene un valor no válido:

```
static void CopyObject(SampleClass original)
{
    if (original == null)
    {
        throw new System.ArgumentException("Parameter cannot be null", "original");
    }
}
```

- Se realiza una llamada inadecuada a un objeto, en función del estado del objeto.

  Un ejemplo podría ser intentar escribir en un archivo de solo lectura. En los casos en los que un estado de objeto no admite una operación, produzca una instancia de InvalidOperationException o un objeto basado en una derivación de esta clase. Este es un ejemplo de un método que genera un objeto InvalidOperationException:

```
class ProgramLog
{
    System.IO.FileStream logFile = null;
    void OpenLog(System.IO.FileInfo fileName, System.IO.FileMode mode) {}

    void WriteLog()
    {
        if (!this.logFile.CanWrite)
        {
            throw new System.InvalidOperationException("Logfile cannot be read-only");
        }
        // Else write data to the log and return.
    }
}
```

- Cuando un argumento de un método genera una excepción.

  En este caso, se debe detectar la excepción original y se debe crear una instancia de ArgumentException. La excepción original debe pasarse al constructor de ArgumentException como el parámetro InnerException:

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        System.ArgumentException argEx = new System.ArgumentException("Index is out of range", "index",
ex);
        throw argEx;
    }
}
```

Las excepciones contienen una propiedad denominada StackTrace. Esta cadena contiene el nombre de los métodos de la pila de llamadas actual, junto con el nombre de archivo y el número de la línea en la que se ha producido la excepción para cada método. Common Language Runtime (CLR) crea automáticamente un objeto StackTrace desde el punto de la instrucción `throw`, de manera que todas las excepciones se deben producir desde el punto en el que debe comenzar el seguimiento de la pila.

Todas las excepciones contienen una propiedad denominada Message. Esta cadena debe establecerse para que explique el motivo de la excepción. Tenga en cuenta que no se debe colocar información confidencial en materia de seguridad en el texto del mensaje. Además de Message, ArgumentException contiene una propiedad denominada ParamName que debe establecerse en el nombre del argumento que ha provocado que se genere la excepción. En el caso de un establecedor de propiedades, ParamName debe establecerse en `value`.

Los métodos públicos y protegidos deben producir excepciones cada vez que no puedan finalizar sus funciones previstas. La clase de excepciones que se produce debe ser la excepción más específica disponible que se ajuste a las condiciones del error. Estas excepciones se deben documentar como parte de la funcionalidad de la clase, y las clases derivadas o actualizaciones de la clase original deben mantener el mismo comportamiento para la compatibilidad con versiones anteriores.

Aspectos que se deben evitar al producir excepciones

En la siguiente lista se identifican los procedimientos que se deben evitar al producir excepciones:

- Las excepciones no deben usarse para cambiar el flujo de un programa como parte de la ejecución normal. Las excepciones solo deben usarse para comunicar y controlar las condiciones de error.
- Las excepciones no se deben devolver como un parámetro o valor devuelto en lugar de producirse.
- No genere System.Exception, System.SystemException, System.NullReferenceException o System.IndexOutOfRangeException de manera intencionada desde su propio código fuente.
- No cree excepciones que se puedan producir en el modo de depuración, pero no en el modo de lanzamiento. Para identificar los errores en tiempo de ejecución durante la fase de desarrollo, use la aserción de depuración.

Definir clases de excepción

Los programas pueden producir una clase de excepción predefinida en el espacio de nombres System (excepto en los casos indicados anteriormente) o crear sus propias clases de excepción mediante la derivación de Exception. Las clases derivadas deben definir al menos cuatro constructores: un constructor sin parámetros, uno que establezca la propiedad de mensaje y otro que establezca las propiedades Message y InnerException. El cuarto constructor se usa para serializar la excepción. Las nuevas clases de excepción deben ser serializables. Por ejemplo:

```
[Serializable()]
public class InvalidDepartmentException : System.Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, System.Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) : base(info, context) { }
}
```

Las nuevas propiedades solo deben agregarse a la clase de excepción cuando los datos que proporcionan son útiles para resolver la excepción. Si se agregan nuevas propiedades a la clase de excepción derivada, se debe invalidar `ToString()` para devolver la información agregada.

Especificación del lenguaje C#

Para obtener más información, vea las secciones Excepciones y La instrucción throw de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Guía de programación de C#
- Excepciones y control de excepciones
- Jerarquía de excepciones
- Control de excepciones

Excepciones generadas por el compilador (Guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

Algunas excepciones las inicia automáticamente el entorno de ejecución .NET cuando se producen errores de operaciones básicas. En la tabla siguiente se enumeran estas excepciones y sus condiciones de error.

EXCEPCIÓN	DESCRIPCIÓN
ArithmeticException	Una clase base para las excepciones que se producen durante las operaciones aritméticas, como DivideByZeroException y OverflowException.
ArrayTypeMismatchException	Se inicia cuando una matriz no puede almacenar un elemento determinado porque el tipo real del elemento es incompatible con el tipo real de la matriz.
DivideByZeroException	Se inicia cuando se intenta dividir un valor entero entre cero.
IndexOutOfRangeException	Se inicia cuando se intenta indexar una matriz y el índice es menor que cero o queda fuera de los límites de la matriz.
InvalidCastException	Se inicia cuando se produce un error en una conversión explícita de un tipo base en una interfaz o un tipo derivado en tiempo de ejecución.
NullReferenceException	Se inicia cuando se intenta hacer referencia a un objeto cuyo valor es null.
OutOfMemoryException	Se inicia cuando se produce un error al intentar asignar memoria con el operador new. Indica que se ha agotado la memoria disponible para el entorno Common Language Runtime.
OverflowException	Se inicia cuando se desborda una operación aritmética en un contexto `checked`.
StackOverflowException	Se inicia cuando se agota la pila de ejecución por tener demasiadas llamadas a métodos pendientes. Normalmente, indica una recursividad muy profunda o infinita.
TypeInitializationException	Se inicia cuando un constructor estático inicia una excepción y no existe una cláusula `catch` compatible para capturarla.

Vea también

- Guía de programación de C#
- Excepciones y control de excepciones
- Control de excepciones
- try-catch

- try-finally
- try-catch-finally

Procedimiento Controlar una excepción mediante Try y Catch (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

El propósito de un bloque try-catch es detectar y controlar una excepción generada por código en funcionamiento. Algunas excepciones se pueden controlar en un bloque `catch` y es posible resolver el problema sin que se vuelva a producir la excepción, pero la mayoría de las veces lo único que se puede hacer es asegurarse de que se produzca la excepción adecuada.

Ejemplo

En este ejemplo, IndexOutOfRangeException no es la excepción más adecuada. Tiene más sentido la excepción ArgumentOutOfRangeException para el método, ya que el error lo provoca el argumento `index` pasado por el autor de la llamada.

```
class TestTryCatch
{
    static int GetInt(int[] array, int index)
    {
        try
        {
            return array[index];
        }
        catch (System.IndexOutOfRangeException e)  // CS0168
        {
            System.Console.WriteLine(e.Message);
            // Set IndexOutOfRangeException to the new exception's InnerException.
            throw new System.ArgumentOutOfRangeException("index parameter is out of range.", e);
        }
    }
}
```

Comentarios

El código que produce una excepción está incluido en el bloque `try`. Se agrega una instrucción `catch` inmediatamente después para controlar `IndexOutOfRangeException`, si se produce. El bloque `catch` controla la excepción `IndexOutOfRangeException` y produce en su lugar la excepción `ArgumentOutOfRangeException`, más adecuada. Para proporcionar al autor de la llamada tanta información como sea posible, considere la posibilidad de especificar la excepción original como InnerException de la nueva excepción. Dado que la propiedad InnerException es readonly, debe asignarla en el constructor de la nueva excepción.

Vea también

- Guía de programación de C#
- Excepciones y control de excepciones
- Control de excepciones

Procedimiento Ejecutar código de limpieza mediante finally (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

El propósito de una instrucción `finally` es asegurarse de que la limpieza necesaria de los objetos, por lo general objetos que contienen recursos externos, se produzca inmediatamente, incluso si se produce una excepción. Un ejemplo de esta limpieza es llamar a Close en un FileStream inmediatamente después de su uso en lugar de esperar a que el objeto sea recolectado por el Common Language Runtime, de esta forma:

```
static void CodeWithoutCleanup()
{
    System.IO.FileStream file = null;
    System.IO.FileInfo fileInfo = new System.IO.FileInfo("C:\\file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Ejemplo

Para activar el código anterior en una instrucción `try-catch-finally`, el código de limpieza se separa del código de trabajo, de esta forma.

```
static void CodeWithCleanup()
{
    System.IO.FileStream file = null;
    System.IO.FileInfo fileInfo = null;

    try
    {
        fileInfo = new System.IO.FileInfo("C:\\file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch(System.UnauthorizedAccessException e)
    {
        System.Console.WriteLine(e.Message);
    }
    finally
    {
        if (file != null)
        {
            file.Close();
        }
    }
}
```

Dado que una excepción puede producirse en cualquier momento dentro del bloque `try` antes de la llamada a `OpenWrite()`, o que podría producirse un error en la propia llamada a `OpenWrite()`, no se garantiza que el archivo está abierto cuando se intenta cerrar. El bloque `finally` agrega una comprobación para asegurarse de que el objeto FileStream no es `null` antes de que pueda llamar al método Close. Sin la comprobación `null`, el bloque

`finally` podría iniciar su propia excepción NullReferenceException, pero debería evitarse generar excepciones en bloques `finally` si es posible.

Una conexión de base de datos es otra buena candidata para cerrarse en un bloque `finally`. Dado que a veces se limita el número de conexiones permitido en un servidor de base de datos, se deben cerrar las conexiones de base de datos tan pronto como sea posible. Si se produce una excepción antes de poder cerrar la conexión, se trata de otro caso en el que usar el bloque `finally` es mejor que esperar a la recolección de elementos no utilizados.

Vea también

- Guía de programación de C#
- Excepciones y control de excepciones
- Control de excepciones
- using (instrucción)
- try-catch
- try-finally
- try-catch-finally

Procedimiento para detectar excepciones no compatibles con CLS

18/03/2020 • 3 minutes to read • Edit Online

Algunos lenguajes. NET, incluido C++/CLI, permiten que los objetos inicien excepciones que no se derivan de Exception. Dichas excepciones se denominan *excepciones de no compatibilidad con CLS* o *no excepciones*. En C# no se pueden producir excepciones de no compatibilidad con CLS, pero se pueden detectar de dos formas:

- Dentro de un bloque `catch (RuntimeWrappedException e)`.

  De forma predeterminada, un ensamblado de Visual C# detecta las excepciones de no compatibilidad con CLS como excepciones ajustadas. Use este método si necesita tener acceso a la excepción original, a la que se puede tener acceso a través de la propiedad RuntimeWrappedException.WrappedException. El procedimiento mostrado más adelante en este tema explica cómo detectar las excepciones de esta manera.

- Dentro de un bloque catch general (un bloque catch sin un tipo de excepción especificado) que se coloca detrás de todos los demás bloques `catch`.

  Use este método cuando quiera realizar alguna acción (como escribir en un archivo de registro) en respuesta a las excepciones de no compatibilidad con CLS y no necesita tener acceso a la información de excepción. De forma predeterminada, el Common Language Runtime ajusta todas las excepciones. Para deshabilitar este comportamiento, agregue este atributo de nivel de ensamblado en el código, normalmente en el archivo AssemblyInfo.cs: `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`.

Para detectar una excepción de no compatibilidad con CLS

Dentro de un bloque `catch(RuntimeWrappedException e)`, se accede a la excepción original mediante la propiedad RuntimeWrappedException.WrappedException.

Ejemplo

En el ejemplo siguiente se muestra cómo detectar una excepción de no compatibilidad con CLS iniciada desde una biblioteca de clases escrita en C++/CLI. Tenga en cuenta que en este ejemplo, el código de cliente de C# conoce por adelantado que el tipo de excepción que se inicia es System.String. Puede convertir la propiedad RuntimeWrappedException.WrappedException de vuelta a su tipo original siempre que el tipo sea accesible desde su código.

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

Vea también

- RuntimeWrappedException
- Excepciones y control de excepciones

Sistema de archivos y el Registro (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

En los artículos siguientes se muestra cómo usar C# y .NET para realizar diversas operaciones básicas en los archivos, las carpetas y el Registro.

En esta sección

TÍTULO	DESCRIPCIÓN
Procedimiento para recorrer en iteración un árbol de directorio	Muestra cómo realizar una iteración manual a través de un árbol de directorio.
Procedimiento para obtener información sobre archivos, carpetas y unidades	Muestra cómo recuperar información como las horas de creación y el tamaño, así como sobre archivos, carpetas y unidades.
Procedimiento para crear archivos o carpetas	Muestra cómo crear un archivo o una carpeta nuevos.
Procedimiento para copiar, eliminar y mover archivos y carpetas (Guía de programación de C#)	Muestra cómo copiar, eliminar y mover archivos y carpetas.
Procedimiento para proporcionar un cuadro de diálogo de progreso para operaciones de archivos	Muestra cómo mostrar un cuadro de diálogo de progreso estándar de Windows para determinadas operaciones de archivo.
Procedimiento para escribir en un archivo texto	Muestra cómo escribir en un archivo de texto.
Procedimiento para leer de un archivo de texto	Muestra cómo leer de un archivo de texto.
Procedimiento para leer un archivo de texto línea a línea	Muestra cómo recuperar el texto de un archivo línea a línea.
Procedimiento para crear una clave en el Registro	Muestra cómo escribir una clave en el registro del sistema.

Secciones relacionadas

- E/S de archivos y secuencias
- Procedimiento para copiar, eliminar y mover archivos y carpetas (Guía de programación de C#)
- Guía de programación de C#
- System.IO

Procedimiento Recorrer en iteración un árbol de directorio (Guía de programación de C#)

29/06/2020 • 11 minutes to read • Edit Online

La frase "recorrer en iteración un árbol de directorios" significa obtener acceso a cada uno de los archivos de todos los subdirectorios anidados bajo una carpeta raíz especificada hasta un nivel de profundidad cualquiera. No es necesario abrir cada archivo. Simplemente puede recuperar el nombre del archivo o subdirectorio como un `string`, o puede recuperar información adicional en el formato de un objeto System.IO.FileInfo o System.IO.DirectoryInfo.

> **NOTE**
> En Windows, los términos "directorio" y "carpeta" se usan indistintamente. La mayor parte de la documentación y del texto de la interfaz de usuario usa el término "carpeta", pero las bibliotecas de clases de .NET usan el término "directorio".

En el caso más simple, en el que sabe con seguridad que tiene permisos de acceso para todos los directorios incluidos en una raíz especificada, puede usar la marca `System.IO.SearchOption.AllDirectories`. Esta marca devuelve todos los subdirectorios anidados que coinciden con el patrón especificado. En el ejemplo siguiente se muestra cómo usar esta marca.

```
root.GetDirectories("*.*", System.IO.SearchOption.AllDirectories);
```

El punto débil de este enfoque es que si uno de los subdirectorios incluidos en la raíz especificada produce una excepción DirectoryNotFoundException o UnauthorizedAccessException, se produce un error en todo el método y no devuelve ningún directorio. Sucede lo mismo cuando usa el método GetFiles. Si tiene que controlar estas excepciones en subcarpetas específicas, debe recorrer manualmente el árbol de directorios, como se muestra en los ejemplos siguientes.

Al recorrer manualmente un árbol de directorios, puede controlar primero los subdirectorios (*recorrido en preorden*) o los archivos (*recorrido en postorden*). Al realizar un recorrido en preorden, se recorre todo el árbol bajo la carpeta actual antes de recorrer en iteración los archivos que están directamente en esa carpeta. Los ejemplos que se proporcionan más adelante en este documento realizan un recorrido en postorden, pero puede modificarlos fácilmente para que realicen un recorrido en preorden.

Otra opción consiste en usar la recursividad o un recorrido basado en la pila. Los ejemplos que se proporcionan más adelante en este documento muestran ambos enfoques.

Si tiene que realizar diversas operaciones en los archivos y las carpetas, puede dividir estos ejemplos en partes mediante la refactorización de la operación en funciones separadas que se puedan invocar usando un solo delegado.

NOTE

Los sistemas de archivos NTFS pueden contener *puntos de reanálisis* en forma de *puntos de unión*, *vínculos simbólicos* y *vínculos físicos*. Los métodos de .NET Framework como GetFiles y GetDirectories no devolverán ningún subdirectorio en un punto de reanálisis. Este comportamiento protege frente al riesgo de provocar un bucle infinito cuando dos puntos de reanálisis se hacen referencia entre sí. En general, debería ser muy cuidadoso al tratar con puntos de reanálisis para asegurarse de no modificar o eliminar archivos involuntariamente. Si quiere obtener un control preciso de los puntos de reanálisis, use la invocación de plataforma o código nativo para llamar directamente a los métodos de sistema de archivos Win32 adecuados.

Ejemplo

En el ejemplo siguiente se muestra cómo recorrer un árbol de directorios mediante recursividad. El enfoque recursivo resulta elegante, pero puede producir una excepción de desbordamiento de la pila si el árbol de directorios es grande y cuenta con muchos elementos anidados.

Las excepciones concretas que se controlan y las acciones determinadas que se realizan en cada archivo o carpeta se proporcionan simplemente como ejemplos. Debe modificar este código para que se ajuste a sus requisitos concretos. Para obtener más información, vea los comentarios del código.

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
System.Collections.Specialized.StringCollection();

    static void Main()
    {
        // Start with drives if you have to search the entire computer.
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read. This
            // is not necessarily the appropriate action in all scenarios.
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read", di.Name);
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
```

```
        try
        {
            files = root.GetFiles("*.*");
        }
        // This is thrown if even one of the files requires permissions greater
        // than the application provides.
        catch (UnauthorizedAccessException e)
        {
            // This code just writes out the message and continues to recurse.
            // You may decide to do something different here. For example, you
            // can try to elevate your privileges and access the file again.
            log.Add(e.Message);
        }

        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }

        if (files != null)
        {
            foreach (System.IO.FileInfo fi in files)
            {
                // In this example, we only access the existing FileInfo object. If we
                // want to open, delete or modify the file, then
                // a try-catch block is required here to handle the case
                // where the file has been deleted since the call to TraverseTree().
                Console.WriteLine(fi.FullName);
            }

            // Now find all the subdirectories under this directory.
            subDirs = root.GetDirectories();

            foreach (System.IO.DirectoryInfo dirInfo in subDirs)
            {
                // Resursive call for each subdirectory.
                WalkDirectoryTree(dirInfo);
            }
        }
    }
}
```

Ejemplo

En el ejemplo siguiente se muestra cómo recorrer en iteración los archivos y las carpetas de un árbol de directorios sin usar la recursividad. Esta técnica usa el tipo de colección genérica Stack<T>, que es una pila de tipo LIFO (último en entrar, primero en salir).

Las excepciones concretas que se controlan y las acciones determinadas que se realizan en cada archivo o carpeta se proporcionan simplemente como ejemplos. Debe modificar este código para que se ajuste a sus requisitos concretos. Para obtener más información, vea los comentarios del código.

```
public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
```

```
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
        Stack<string> dirs = new Stack<string>(20);

        if (!System.IO.Directory.Exists(root))
        {
            throw new ArgumentException();
        }
        dirs.Push(root);

        while (dirs.Count > 0)
        {
            string currentDir = dirs.Pop();
            string[] subDirs;
            try
            {
                subDirs = System.IO.Directory.GetDirectories(currentDir);
            }
            // An UnauthorizedAccessException exception will be thrown if we do not have
            // discovery permission on a folder or file. It may or may not be acceptable
            // to ignore the exception and continue enumerating the remaining files and
            // folders. It is also possible (but unlikely) that a DirectoryNotFound exception
            // will be raised. This will happen if currentDir has been deleted by
            // another application or thread after our call to Directory.Exists. The
            // choice of which exceptions to catch depends entirely on the specific task
            // you are intending to perform and also on how much you know with certainty
            // about the systems on which this code will run.
            catch (UnauthorizedAccessException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }

            string[] files = null;
            try
            {
                files = System.IO.Directory.GetFiles(currentDir);
            }

            catch (UnauthorizedAccessException e)
            {

                Console.WriteLine(e.Message);
                continue;
            }

            catch (System.IO.DirectoryNotFoundException e)
            {
                Console.WriteLine(e.Message);
                continue;
            }
            // Perform the required action on each file here.
            // Modify this block to perform your required task.
            foreach (string file in files)
            {
                try
                {
                    // Perform whatever action is required in your scenario.
                    System.IO.FileInfo fi = new System.IO.FileInfo(file);
                    Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
                }
                catch (System.IO.FileNotFoundException e)
                {
```

```
                // If file was deleted by a separate application
                //  or thread since the call to TraverseTree()
                // then just continue.
                Console.WriteLine(e.Message);
                continue;
            }
        }

        // Push the subdirectories onto the stack for traversal.
        // This could also be done before handing the files.
        foreach (string str in subDirs)
            dirs.Push(str);
    }
  }
}
```

Generalmente se tarda mucho tiempo en comprobar cada carpeta para determinar si su aplicación tiene permiso para abrirla. Por consiguiente, el ejemplo de código incluye esa parte de la operación en un bloque `try/catch`. Puede modificar el bloque `catch` de manera que, cuando se le deniegue el acceso a una carpeta, intente elevar sus permisos y obtener acceso a esta de nuevo. Como norma, detecte solamente las excepciones que puede controlar sin dejar la aplicación en un estado desconocido.

Si debe almacenar el contenido de un árbol de directorios, ya sea en memoria o en el disco, la mejor opción es almacenar solamente la propiedad FullName (de tipo `string`) para cada archivo. Después, puede usar esta cadena para crear un nuevo objeto FileInfo o DirectoryInfo, según sea necesario, o para abrir cualquier archivo que requiera un procesamiento adicional.

Programación sólida

Un código eficaz de iteración de archivos debe tener en cuenta las numerosas dificultades del sistema de archivos. Para más información sobre el sistema de archivos de Windows, vea NTFS overview (Introducción a NTFS).

Vea también

- System.IO
- LINQ y directorios de archivos
- Registro y sistema de archivos (Guía de programación de C#)

Procedimiento Obtener información sobre archivos, carpetas y unidades (Guía de programación de C#)

29/06/2020 • 4 minutes to read • Edit Online

En .NET, puede acceder a información del sistema de archivos mediante las clases siguientes:

- System.IO.FileInfo
- System.IO.DirectoryInfo
- System.IO.DriveInfo
- System.IO.Directory
- System.IO.File

Las clases FileInfo y DirectoryInfo representan un archivo o directorio y contienen propiedades que exponen muchos de los atributos de archivo admitidos por el sistema de archivos NTFS. También contienen métodos para abrir, cerrar, mover y eliminar archivos y carpetas. Para crear instancias de estas clases, pase una cadena que represente el nombre del archivo, carpeta o unidad en el constructor:

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

También puede obtener los nombres de archivos, carpetas o unidades mediante llamadas a DirectoryInfo.GetDirectories, DirectoryInfo.GetFiles y DriveInfo.RootDirectory.

Las clases System.IO.Directory y System.IO.File proporcionan métodos estáticos para recuperar información sobre directorios y archivos.

Ejemplo

En el ejemplo siguiente se muestran diversas maneras de obtener acceso a información sobre archivos y carpetas.

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");

        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }

        // Get the subdirectories directly that is under the root.
```

```
        // See "How to: Iterate Through a Directory Tree" for an example of how to
        // iterate through an entire tree.
        System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories("*.*");

        foreach (System.IO.DirectoryInfo d in dirInfos)
        {
            Console.WriteLine(d.Name);
        }

        // The Directory and File classes provide several static methods
        // for accessing files and directories.

        // Get the current application directory.
        string currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Get an array of file names as strings rather than FileInfo objects.
        // Use this method when storage space is an issue, and when you might
        // hold on to the file name reference for a while before you try to access
        // the file.
        string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

        foreach (string s in files)
        {
            // Create the FileInfo object only when needed to ensure
            // the information is as current as possible.
            System.IO.FileInfo fi = null;
            try
            {
                 fi = new System.IO.FileInfo(s);
            }
            catch (System.IO.FileNotFoundException e)
            {
                // To inform the user and continue is
                // sufficient for this demonstration.
                // Your application may require different behavior.
                Console.WriteLine(e.Message);
                continue;
            }
            Console.WriteLine("{0} : {1}",fi.Name, fi.Directory);
        }

        // Change the directory. In this case, first check to see
        // whether it already exists, and create it if it does not.
        // If this is not appropriate for your application, you can
        // handle the System.IO.IOException that will be raised if the
        // directory cannot be found.
        if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\"))
        {
            System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");
        }

        System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

        currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Programación sólida

Al procesar cadenas de ruta de acceso especificadas por el usuario, también debe controlar las excepciones para las

condiciones siguientes:

- El nombre del archivo es incorrecto. Por ejemplo, contiene caracteres no válidos o solo espacios en blanco.
- El nombre del archivo es nulo.
- La longitud del nombre de archivo es superior a la longitud máxima definida por el sistema.
- El nombre de archivo contiene un signo de dos puntos (:).

Si la aplicación no tiene permisos suficientes para leer el archivo especificado, el método `Exists` devuelve `false` con independencia de si existe una ruta de acceso; el método no produce una excepción.

Vea también

- System.IO
- Guía de programación de C#
- Registro y sistema de archivos (Guía de programación de C#)

Procedimiento Crear archivos o carpetas (Guía de programación de C#)

29/06/2020 • 5 minutes to read • Edit Online

Puede crear una carpeta en el equipo mediante programación, crear una subcarpeta, crear un archivo en la subcarpeta y escribir datos en el archivo.

Ejemplo

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
```

```
            }
        }
        else
        {
            Console.WriteLine("File \"{0}\" already exists.", fileName);
            return;
        }

        // Read and display the data from your file.
        try
        {
            byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
            foreach (byte b in readBuffer)
            {
                Console.Write(b + " ");
            }
            Console.WriteLine();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
    // Sample output:

    // Path to my file: c:\Top-Level Folder\SubFolder\ttxvauxe.vv0

    //0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
    //30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
    // 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 8
    //3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}
```

Si la carpeta ya existe, CreateDirectory no efectúa ninguna acción y no se devuelve ninguna excepción. Sin embargo, File.Create reemplaza un archivo existente por otro nuevo. En el ejemplo se usa una instrucción `if`-`else` para evitar que se sustituya un archivo existente.

Al realizar los siguientes cambios en el ejemplo, puede especificar diferentes resultados en función de si ya existe un archivo con un nombre determinado. Si no existe ese archivo, el código crea uno. Si ese archivo ya existe, el código anexa datos a ese archivo.

- Especifique un nombre de archivo no aleatorio.

```
// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";
```

- Sustituya la instrucción `if`-`else` por la instrucción `using` en el código siguiente.

```
using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}
```

Ejecute el ejemplo varias veces para comprobar que los datos se agreguen al archivo cada vez.

Para obtener más valores `FileMode` que puede probar, consulte FileMode.

Las condiciones siguientes pueden provocar una excepción:

- El nombre de la carpeta tiene un formato incorrecto. Por ejemplo, contiene caracteres no válidos o solo tiene espacios en blanco (clase ArgumentException). Use la clase Path para crear nombres de ruta válidos.
- La carpeta principal de la que se va a crear es de solo lectura (clase IOException).
- El nombre de la carpeta es `null` (clase ArgumentNullException).
- El nombre de la carpeta es demasiado largo (clase PathTooLongException).
- El nombre de la carpeta es solo un carácter de dos puntos, ":" (clase PathTooLongException).

Seguridad de .NET

En los casos de confiabilidad parcial, es posible que se devuelva una instancia de la clase SecurityException.

Si no tiene permiso para crear la carpeta, el ejemplo devuelve una instancia de la clase UnauthorizedAccessException.

Vea también

- System.IO
- Guía de programación de C#
- Registro y sistema de archivos (Guía de programación de C#)

Procedimiento Copiar, eliminar y mover archivos y carpetas (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

En los siguientes ejemplos se muestra cómo copiar, mover y eliminar archivos y carpetas de una manera sincrónica con las clases System.IO.File, System.IO.Directory, System.IO.FileInfo y System.IO.DirectoryInfo desde el espacio de nombres System.IO. En estos ejemplos no se proporciona una barra de progreso ni ninguna otra interfaz de usuario. Si quiere proporcionar un cuadro de diálogo de progreso estándar, consulte Procedimiento Proporcionar un cuadro de diálogo de progreso para operaciones de archivos.

Use System.IO.FileSystemWatcher para proporcionar eventos que le permitan calcular el progreso al realizar operaciones en varios archivos. Otro enfoque consiste en usar la invocación de plataforma para llamar a los métodos pertinentes relacionados con archivos en el shell de Windows. Para obtener información sobre cómo realizar estas operaciones de archivo de forma asincrónica, vea E/S de archivos asincrónica.

Ejemplo

En el ejemplo siguiente se muestra cómo copiar archivos y directorios.

```
// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and files:
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath =  @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);
        string destFile = System.IO.Path.Combine(targetPath, fileName);

        // To copy a folder's contents to a new location:
        // Create a new target folder.
        // If the directory already exists, this method does not create a new directory.
        System.IO.Directory.CreateDirectory(targetPath);

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);

        // To copy all the files in one directory to another directory.
        // Get the files in the source folder. (To recursively iterate through
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //       in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePath);

            // Copy the files and overwrite destination files if they already exist.
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file name from the path.
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileName);
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Ejemplo

En el ejemplo siguiente se muestra cómo mover archivos y directorios.

```
// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}
```

Ejemplo

En el ejemplo siguiente se muestra cómo eliminar archivos y directorios.

```
// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
```

```
        {
            Console.WriteLine(e.Message);
        }
        // Delete a directory and all subdirectories with Directory static method...
        if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))
        {
            try
            {
                System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);
            }

            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
            }
        }

        // ...or with DirectoryInfo instance method.
        System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");
        // Delete this dir and all subdirs.
        try
        {
            di.Delete(true);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

Vea también

- System.IO
- Guía de programación de C#
- Registro y sistema de archivos (Guía de programación de C#)
- Procedimiento para proporcionar un cuadro de diálogo de progreso para operaciones de archivos
- E/S de archivos y secuencias
- Tareas de E/S comunes

Procedimiento Proporcionar un cuadro de diálogo de progreso para operaciones de archivos (Guía de programación de C#)

18/03/2020 • 2 minutes to read • Edit Online

Puede proporcionar un cuadro de diálogo estándar que muestra el progreso en las operaciones de archivo en Windows si usa el método CopyFile(String, String, UIOption) en el espacio de nombres Microsoft.VisualBasic.

> **NOTE**
>
> Es posible que el equipo muestre nombres o ubicaciones diferentes para algunos de los elementos de la interfaz de usuario de Visual Studio en las siguientes instrucciones. La edición de Visual Studio que se tenga y la configuración que se utilice determinan estos elementos. Para obtener más información, vea Personalizar el IDE.

Para agregar una referencia en Visual Studio

1. En la barra de menús, elija **Proyecto**, **Agregar referencia**.

   Aparecerá el cuadro de diálogo **Administrador de referencias**.

2. En el área **Ensamblados**, elija **Framework** si no está ya seleccionado.

3. En la lista de nombres, seleccione la casilla **Microsoft.VisualBasic** y elija el botón **Aceptar** para cerrar el cuadro de diálogo.

Ejemplo

El siguiente código copia el directorio que especifica `sourcePath` en el directorio que especifica `destinationPath`. Este código también proporciona un cuadro de diálogo estándar en el que se muestra el tiempo estimado restante antes de que finalice la operación.

```
// The following using directive requires a project reference to Microsoft.VisualBasic.
using Microsoft.VisualBasic.FileIO;

class FileProgress
{
    static void Main()
    {
        // Specify the path to a folder that you want to copy. If the folder is small,
        // you won't have time to see the progress dialog box.
        string sourcePath = @"C:\Windows\symbols\";
        // Choose a destination for the copied files.
        string destinationPath = @"C:\TestFolder";

        FileSystem.CopyDirectory(sourcePath, destinationPath,
            UIOption.AllDialogs);
    }
}
```

Vea también

- Registro y sistema de archivos (Guía de programación de C#)

Procedimiento Escribir en un archivo de texto (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

En estos ejemplos se muestran varias formas de escribir texto en un archivo. Los dos primeros ejemplos usan métodos estáticos convenientes en la clase System.IO.File para escribir cada elemento de cualquier `IEnumerable<string>` y una cadena en un archivo de texto. En el ejemplo 3 se muestra cómo agregar texto a un archivo cuando es necesario procesar cada línea individualmente a medida que se escribe en el archivo. Los ejemplos 1-3 sobrescriben todo el contenido del archivo, pero el ejemplo 4 muestra cómo anexar texto a un archivo existente.

Todos estos ejemplos escriben literales de cadena en los archivos. Si quiere aplicar formato al texto escrito en un archivo, use el método Format o la característica interpolación de cadenas de C#.

Ejemplo

```
class WriteTextFile
{
    static void Main()
    {

        // These examples assume a "C:\Users\Public\TestFolder" folder on your machine.
        // You can modify the path if necessary.

        // Example #1: Write an array of strings to a file.
        // Create a string array that consists of three lines.
        string[] lines = { "First line", "Second line", "Third line" };
        // WriteAllLines creates a file, writes a collection of strings to the file,
        // and then closes the file.  You do NOT need to call Flush() or Close().
        System.IO.File.WriteAllLines(@"C:\Users\Public\TestFolder\WriteLines.txt", lines);

        // Example #2: Write one string to a text file.
        string text = "A class is the most powerful data type in C#. Like a structure, " +
                      "a class defines the data and behavior of the data type. ";
        // WriteAllText creates a file, writes the specified string to the file,
        // and then closes the file.    You do NOT need to call Flush() or Close().
        System.IO.File.WriteAllText(@"C:\Users\Public\TestFolder\WriteText.txt", text);

        // Example #3: Write only some strings in an array to a file.
        // The using statement automatically flushes AND CLOSES the stream and calls
        // IDisposable.Dispose on the stream object.
        // NOTE: do not use FileStream for text files because it writes bytes, but StreamWriter
        // encodes the output as text.
        using (System.IO.StreamWriter file =
            new System.IO.StreamWriter(@"C:\Users\Public\TestFolder\WriteLines2.txt"))
        {
            foreach (string line in lines)
            {
                // If the line doesn't contain the word 'Second', write the line to the file.
                if (!line.Contains("Second"))
                {
                    file.WriteLine(line);
                }
            }
        }

        // Example #4: Append new text to an existing file.
        // The using statement automatically flushes AND CLOSES the stream and calls
```

```
        // IDisposable.Dispose on the stream object.
        using (System.IO.StreamWriter file =
            new System.IO.StreamWriter(@"C:\Users\Public\TestFolder\WriteLines2.txt", true))
        {
            file.WriteLine("Fourth line");
        }
    }
}
 //Output (to WriteLines.txt):
 //   First line
 //   Second line
 //   Third line

 //Output (to WriteText.txt):
 //   A class is the most powerful data type in C#. Like a structure, a class defines the data and behavior of
the data type.

 //Output to WriteLines2.txt after Example #3:
 //   First line
 //   Third line

 //Output to WriteLines2.txt after Example #4:
 //   First line
 //   Third line
 //   Fourth line
```

Programación sólida

Las condiciones siguientes pueden provocar una excepción:

- El archivo ya existe y es de solo lectura.
- Puede que el nombre de ruta de acceso sea demasiado largo.
- Es posible que el disco esté lleno.

Vea también

- Guía de programación de C#
- Registro y sistema de archivos (Guía de programación de C#)
- Ejemplo: guardar una colección en el almacenamiento para la aplicación

Procedimiento Leer de un archivo de texto (Guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

En este ejemplo se lee el contenido de un archivo de texto con los métodos estáticos ReadAllText y ReadAllLines de la clase System.IO.File.

Para obtener un ejemplo en el que se usa StreamReader, consulte Procedimiento Leer un archivo de texto línea a línea.

> **NOTE**
> Los archivos que se usan en este ejemplo se crean en el tema Procedimiento Escribir en un archivo texto.

Ejemplo

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Compilar el código

Copie el código y péguelo en una aplicación de consola de C#.

Si no está usando los archivos de texto de Procedimiento Escribir en un archivo texto, reemplace el argumento a

`ReadAllText` y a `ReadAllLines` por la ruta de acceso adecuada y el nombre de archivo en el equipo.

Programación sólida

Las condiciones siguientes pueden provocar una excepción:

- El archivo no existe o no existe en la ubicación especificada. Compruebe la ruta de acceso y la ortografía del nombre de archivo.

Seguridad de .NET

No confíe en el nombre de un archivo para determinar el contenido del archivo. Por ejemplo, el archivo `myFile.cs` puede que no sea un archivo de código fuente de C#.

Vea también

- System.IO
- Guía de programación de C#
- Registro y sistema de archivos (Guía de programación de C#)

Procedimiento para leer un archivo de texto de línea en línea (guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

En este ejemplo se lee el contenido de un archivo de texto línea a línea en una cadena mediante el método `ReadLine` de la clase `StreamReader`. Cada línea de texto se almacena en la cadena `line` y se muestra en la pantalla.

Ejemplo

```
int counter = 0;
string line;

// Read the file and display it line by line.
System.IO.StreamReader file =
    new System.IO.StreamReader(@"c:\test.txt");
while((line = file.ReadLine()) != null)
{
    System.Console.WriteLine(line);
    counter++;
}

file.Close();
System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

Compilar el código

Copie el código y péguelo en el método `Main` de una aplicación de consola.

Reemplace `"c:\test.txt"` por el nombre de archivo real.

Programación sólida

Las condiciones siguientes pueden provocar una excepción:

- El archivo podría no existir.

Seguridad de .NET

No tome ninguna decisión sobre el contenido del archivo basándose en su nombre. Por ejemplo, es posible que el archivo `myFile.cs` no sea un archivo de código fuente de C#.

Vea también

- System.IO
- Guía de programación de C#
- Registro y sistema de archivos (Guía de programación de C#)

Procedimiento para crear una clave del Registro (guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

En este ejemplo se agrega el par de valores "Name" e "Isabella" al Registro del usuario actual en la clave "Names".

Ejemplo

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

Compilar el código

- Copie el código y péguelo en el método `Main` de una aplicación de consola.
- Sustituya el parámetro `Names` por el nombre de una clave que exista directamente en el nodo HKEY_CURRENT_USER del Registro.
- Sustituya el parámetro `Name` por el nombre de un valor que exista directamente en el nodo Names.

Programación sólida

Examine la estructura del Registro para buscar una ubicación adecuada para la clave. Por ejemplo, es posible que quiera abrir la clave Software del usuario actual y crear una clave con el nombre de la empresa. Luego agregue los valores del Registro a la clave de la empresa.

Las condiciones siguientes pueden generar una excepción:

- Que el nombre de la clave sea nulo.
- Que el usuario no tenga permisos para crear claves del Registro.
- Que el nombre de la clave supere el límite de 255 caracteres.
- Que la clave esté cerrada.
- Que la clave del Registro sea de solo lectura.

Seguridad de .NET

Es más seguro escribir datos en la carpeta de usuario (`Microsoft.Win32.Registry.CurrentUser`) que en el equipo local (`Microsoft.Win32.Registry.LocalMachine`).

Cuando se crea un valor del Registro, se debe decidir qué hacer si ese valor ya existe. Puede que otro proceso, quizás uno malintencionado, ya haya creado el valor y tenga acceso a él. Al colocar datos en el valor del Registro, estos están a disposición del otro proceso. Para evitarlo, use el método `Overload:Microsoft.Win32.RegistryKey.GetValue` . Si la clave aún no existe, devuelve null.

No es seguro almacenar secretos, como contraseñas, en el Registro como texto sin formato, aunque la clave del Registro esté protegida mediante listas de control de acceso (ACL).

Vea también

- System.IO
- Guía de programación de C#
- Registro y sistema de archivos (Guía de programación de C#)
- Read, write and delete from the registry with C# (Leer, escribir y eliminar en el Registro con C#)

Interoperabilidad (Guía de programación de C#)

29/06/2020 • 2 minutes to read • Edit Online

La interoperabilidad permite conservar y aprovechar las inversiones existentes en código no administrado. El código que se ejecuta bajo el control de Common Language Runtime (CLR) se denomina *código administrado*, y el código que se ejecuta fuera de CLR se denomina *código no administrado*. COM, COM+, los componentes de C++, los componentes de ActiveX y la API Windows de Microsoft son ejemplos de código no administrado.

.NET habilita la interoperabilidad con código no administrado a través de los servicios de invocación de plataforma, el espacio de nombres System.Runtime.InteropServices, la interoperabilidad de C++ y la interoperabilidad COM.

En esta sección

Información general sobre interoperabilidad
Se describen métodos para habilitar la interoperabilidad entre el código administrado y el código no administrado de C#.

Procedimiento para acceder a objetos de interoperabilidad de Office mediante características de C#
Describe las características introducidas en Visual C# para facilitar la programación de Office.

Procedimiento para usar propiedades indizadas en la programación de interoperabilidad COM
Se describe cómo utilizar las propiedades indizadas para acceder a las propiedades de COM que tienen parámetros.

Procedimiento para usar la invocación de plataforma para reproducir un archivo WAV
Se describe cómo usar los servicios de invocación de plataforma para reproducir un archivo de sonido .wav en el sistema operativo Windows.

Tutorial: Programación de Office
Muestra cómo crear un libro de Excel y un documento de Word que contiene un vínculo al libro.

Clase COM de ejemplo
Muestra cómo exponer una clase de C# como un objeto COM.

Especificación del lenguaje C#

Para obtener más información, consulte la sección Conceptos básicos de Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Marshal.ReleaseComObject
- Guía de programación de C#
- Interoperar con código no administrado
- Tutorial: Programación de Office

Información general sobre interoperabilidad (Guía de programación de C#)

29/06/2020 • 6 minutes to read • Edit Online

En el tema se describen métodos para habilitar la interoperabilidad entre el código administrado y el código no administrado de C#.

Invocación de plataforma

La *invocación de plataforma* es un servicio que permite al código administrado llamar a funciones no administradas que se implementan en bibliotecas de vínculos dinámicos (DLL), como las de la API de Microsoft Windows. Busca y llama a una función exportada y calcula las referencias de sus argumentos (enteros, cadenas, matrices, estructuras etc.) a través de los límites de interoperación según sea necesario.

Para más información, consulte Consumir funciones DLL no administradas y Procedimiento Utilizar la invocación de plataforma para reproducir un archivo de sonido.

> **NOTE**
>
> Common Language Runtime (CLR) administra el acceso a los recursos del sistema. Si se llama al código no administrado que está fuera de CLR, se omite este mecanismo de seguridad y, por lo tanto, existe un riesgo de seguridad. Por ejemplo, el código no administrado podría llamar directamente a recursos en código no administrado, omitiendo los mecanismos de seguridad de CLR. Para obtener más información, vea Seguridad en .NET.

Interoperabilidad de C++

Puede usar la interoperabilidad de C++, también conocida como It Just Works (IJW), para encapsular una clase de C++ nativa de modo que el código creado en C# o en otro lenguaje de .NET pueda consumirla. Para ello, escriba código de C++ para encapsular un componente DLL o COM nativo. A diferencia de otros lenguajes de .NET, Visual C++ cuenta con compatibilidad de interoperabilidad que permite que haya código administrado y no administrado en la misma aplicación, e incluso en el mismo archivo. Después, compile el código de C++ mediante el modificador del compilador **/clr** para generar un ensamblado administrado. Finalmente, agregue una referencia al ensamblado en el proyecto de C# y use los objetos encapsulados igual que usaría otras clases administradas.

Exponer componentes COM en C#

Puede usar un componente COM de un proyecto de C#. Los pasos generales son los siguientes:

1. Busque un componente COM para usarlo y registrarlo. Use regsvr32.exe para registrar un componente DLL COM o para anular su registro.

2. Agregue al proyecto una referencia a la biblioteca de tipos o al componente COM.

   Al agregar la referencia, Visual Studio usa Tlbimp.exe (importador de la biblioteca de tipos), que toma una biblioteca de tipos como entrada para generar un ensamblado de interoperabilidad de .NET. El ensamblado, también denominado "contenedor RCW", contiene las clases e interfaces administradas que encapsulan las interfaces y clases COM que se encuentran en la biblioteca de tipos. Visual Studio agrega al proyecto una referencia al ensamblado generado.

3. Cree una instancia de una clase definida en el RCW. Este, a su vez, crea una instancia del objeto COM.

4. Use el objeto igual que usa otros objetos administrados. Cuando el objeto sea reclamado por la recolección de elementos no utilizados, la instancia del objeto COM también se liberará de la memoria.

Para obtener más información, vea Exponer componentes COM en .NET Framework.

Exponer C# en COM

Los clientes COM pueden consumir tipos de C# que se han expuesto correctamente. Los pasos básicos para exponer tipos de C# son los siguientes:

1. Agregue atributos de interoperabilidad al proyecto de C#.

   Puede hacer que un ensamblado COM sea visible al modificar las propiedades del proyecto de Visual C#. Para obtener más información, vea Información de ensamblado (Cuadro de diálogo).

2. Genere una biblioteca de tipos COM y regístrela para el uso de COM.

   Puede modificar las propiedades del proyecto de Visual C# para registrar automáticamente el ensamblado de C# para la interoperabilidad COM. Visual Studio usa Regasm.exe (herramienta de registro de ensamblados), con el modificador de la línea de comandos `/tlb`, que toma un ensamblado administrado como entrada, para generar una biblioteca de tipos. Esta biblioteca de tipos describe los tipos `public` del ensamblado y agrega entradas del registro para que los clientes COM puedan crear clases administradas.

Para obtener más información, vea Exponer componentes de .NET Framework en COM y Clase COM de ejemplo.

Vea también

- Improving Interop Performance (Mejorar el rendimiento interoperativo)
- Introducción a la interoperabilidad entre COM y .NET
- Información general sobre la interoperabilidad COM (Visual Basic)
- Serialización de interoperabilidad
- Interoperating with Unmanaged Code (Interoperar con código no administrado)
- Guía de programación de C#

Procedimiento Tener acceso a objetos de interoperabilidad de Office mediante las características de Visual C# (Guía de programación de C#)

18/03/2020 • 20 minutes to read • Edit Online

C# tiene nuevas características que simplifican el acceso a objetos de la API de Office. Las nuevas características incluyen argumentos con nombre y opcionales, un nuevo tipo llamado `dynamic` y la capacidad de pasar argumentos a parámetros de referencia en los métodos COM como si fueran parámetros de valor.

En este tema se utilizarán las nuevas características para escribir código que crea y muestra una hoja de cálculo de Microsoft Office Excel. A continuación, se escribirá código para agregar un documento de Office Word que contiene un icono que está vinculado a la hoja de cálculo de Excel.

Para completar este tutorial, es necesario tener Microsoft Office Excel 2007 y Microsoft Office Word 2007 —o una versión posterior— instalados en el equipo.

> **NOTE**
> Es posible que el equipo muestre nombres o ubicaciones diferentes para algunos de los elementos de la interfaz de usuario de Visual Studio en las siguientes instrucciones. La edición de Visual Studio que se tenga y la configuración que se utilice determinan estos elementos. Para obtener más información, vea Personalizar el IDE.

Para crear una aplicación de consola nueva

1. Inicie Visual Studio.
2. En el menú **Archivo** , elija **Nuevo**y haga clic en **Proyecto**. Aparecerá el cuadro de diálogo **Nuevo proyecto** .
3. En el panel **Plantillas instaladas**, expanda **Visual C#** y haga clic en **Windows**.
4. En la parte superior del cuadro de diálogo **Nuevo proyecto**, compruebe si **.NET Framework 4** (o una versión posterior) está seleccionado como una plataforma de destino.
5. En el panel **Plantillas**, haga clic en **Aplicación de consola**.
6. Escriba un nombre para el proyecto en el campo **Nombre**.
7. Haga clic en **Aceptar**.

   El proyecto nuevo aparece en el **Explorador de soluciones**.

Para agregar referencias

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el nombre del proyecto y luego haga clic en **Agregar referencia**. Aparecerá el cuadro de diálogo **Agregar referencia**.
2. En la página de **Ensamblados**, seleccione **Microsoft.Office.Interop.Word** en la lista **Nombre de componente** y, después, mantenga presionada la tecla CTRL y seleccione

Microsoft.Office.Interop.Excel. Si no ve los ensamblados, asegúrese de que están instalados y que se muestran. Vea Cómo: Instalación de ensamblados de interoperabilidad primarios de Office.

3. Haga clic en **Aceptar**.

Para agregar las directivas using necesarias

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el archivo *Program.cs* y, después, haga clic en **Ver código**.

2. Agregue las directivas `using` siguientes a la parte superior del archivo de código:

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Para crear una lista de las cuentas bancarias

1. Pegue la definición de clase siguiente en **Program.cs**, bajo la clase `Program`.

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. Agregue el código siguiente al método `Main` para crear una lista `bankAccounts` lista que contenga dos cuentas.

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

Para declarar un método que exporta información de cuentas a Excel

1. Agregue el método siguiente a la clase `Program` para configurar una hoja de cálculo de Excel.

   El método Add tiene un parámetro opcional para especificar una plantilla determinada. Los parámetros opcionales introducidos en C# 4 permiten omitir el argumento para ese parámetro si se desea utilizar el valor predeterminado del parámetro. Dado que en el código siguiente no se envía ningún argumento, `Add` usa la plantilla predeterminada y crea un libro nuevo. La instrucción equivalente en versiones anteriores de C# requiere un argumento de marcador de posición: `ExcelApp.Workbooks.Add(Type.Missing)`.

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a praticular template.
    // Because no argument is sent in this example, Add creates a new workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

2. Agregue el siguiente código al final de `DisplayInExcel`. El código inserta valores en las dos primeras columnas de la primera fila de la hoja de cálculo.

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

3. Agregue el siguiente código al final de `DisplayInExcel`. El bucle `foreach` coloca la información de la lista de cuentas en las dos primeras columnas de filas sucesivas de la hoja de cálculo.

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

4. Agregue el código siguiente al final de `DisplayInExcel` para ajustar los anchos de columna a fin de adaptarlos al contenido.

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

Las versiones anteriores de C# requieren una conversión explícita para estas operaciones, ya que `ExcelApp.Columns[1]` devuelve `Object` y `AutoFit` es un método Range de Excel. Las siguientes líneas muestran la conversión.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C# 4, y versiones posteriores, convierte automáticamente el valor `Object` devuelto en `dynamic` si se hace referencia al ensamblado mediante la opción del compilador -link o, de forma equivalente, si la propiedad **Incrustar tipos de interoperabilidad** de Excel se establece en true. El valor predeterminado de esta propiedad es true.

Para ejecutar el proyecto

1. Agregue la línea siguiente al final de `Main`.

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

2. Presione CTRL+F5.

   Aparece una hoja de cálculo de Excel que contiene los datos de las dos cuentas.

Para agregar un documento de Word

1. Para ilustrar las formas adicionales en que C# 4, y versiones posteriores, mejoran la programación de Office, el código siguiente abre una aplicación de Word y crea un icono que se vincula a la hoja de cálculo de Excel.

   Pegue el método `CreateIconInWordDoc`, proporcionado más adelante en este paso, en la clase `Program`. `CreateIconInWordDoc` usa argumentos con nombre y opcionales para reducir la complejidad de las llamadas de método a Add y PasteSpecial. Estas llamadas incorporan otras dos nuevas características introducidas en C# 4 que simplifican las llamadas a métodos COM que tienen parámetros de referencia. En primer lugar, puede enviar argumentos a los parámetros de referencia como si fueran parámetros de valor. Es decir, puede enviar valores directamente, sin necesidad de crear una variable para cada parámetro de referencia. El compilador genera variables temporales para contener los valores de argumento y las descarta cuando se regresa de la llamada. En segundo lugar, se puede omitir la palabra clave `ref` en la lista de argumentos.

   El método `Add` tiene cuatro parámetros de referencia, todos ellos opcionales. En C# 4.0 y versiones posteriores, puede omitir los argumentos de cualquiera o de todos los parámetros si quiere usar sus valores predeterminados. En C# 3.0 y versiones anteriores, se debe proporcionar un argumento para cada parámetro; el argumento debe ser una variable, ya que los parámetros son parámetros de referencia.

   El método `PasteSpecial` inserta el contenido del Portapapeles. El método tiene siete parámetros de referencia, todos ellos opcionales. El siguiente código especifica los argumentos para dos de ellos: `Link`, para crear un vínculo con el origen del contenido del Portapapeles, y `DisplayAsIcon`, para mostrar el vínculo como un icono. En C# 4.0 y versiones posteriores, puede usar argumentos con nombre para esos dos y omitir los demás. Aunque se trata de parámetros de referencia, no es necesario utilizar la palabra clave `ref` ni crear variables para enviarlas como argumentos. Puede enviar los valores directamente. En C# 3.0 y versiones anteriores, debe proporcionar un argumento de variable para cada parámetro de referencia.

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

En C# 3.0 y versiones anteriores del lenguaje, es necesario el código siguiente, que es más complejo.

```
static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to contruct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
                                    ref link,
                                    ref useDefaultValue,
                                    ref displayAsIcon,
                                    ref useDefaultValue,
                                    ref useDefaultValue,
                                    ref useDefaultValue);
}
```

2. Agregue la siguiente instrucción al final de `Main`.

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

3. Agregue la siguiente instrucción al final de `DisplayInExcel`. El método `Copy` agrega la hoja de cálculo en el Portapapeles.

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

4. Presione CTRL+F5.

   Un documento de Word aparecerá con un icono. Haga doble clic en el icono para abrir la hoja de cálculo en primer plano.

Para establecer la propiedad Incrustar tipos de interoperabilidad

1. Es posible realizar mejoras adicionales si se llama a un tipo COM que no requiere un ensamblado de interoperabilidad primario (PIA) en tiempo de ejecución. Si se elimina la dependencia de PIA, la versión gana en independencia y se facilita la implementación. Para obtener más información sobre las ventajas de la programación sin PIA, consulte Tutorial: Insertar los tipos de los ensamblados administrados.

Además, programar resulta más fácil porque los tipos requeridos y devueltos por los métodos COM se pueden representar con el tipo `dynamic` en vez de `Object`. Las variables de tipo `dynamic` no se evalúan hasta el tiempo de ejecución, lo que elimina la necesidad de la conversión explícita. Para obtener más información, vea Uso del tipo dynamic.

En C# 4, el comportamiento predeterminado es insertar información de tipos en lugar de utilizar los PIA. Debido a ese comportamiento predeterminado, algunos de los ejemplos anteriores se simplifican dado que no es necesaria la conversión explícita. Por ejemplo, la declaración de `worksheet` en `DisplayInExcel` se escribe como `Excel._Worksheet workSheet = excelApp.ActiveSheet` en lugar de `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. Las llamadas a `AutoFit` en el mismo método también requerirían conversión explícita sin el valor predeterminado, porque `ExcelApp.Columns[1]` devuelve un `Object` y `AutoFit` es un método de Excel. En el código siguiente se muestra la conversión.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. Para cambiar el valor predeterminado y usar los PIA en lugar de insertar información de tipos, expanda el nodo **Referencias** del **Explorador de soluciones** y, después, seleccione **Microsoft.Office.Interop.Excel** o **Microsoft.Office.Interop.Word**.

3. Si no ve la ventana **Propiedades**, presione **F4**.

4. Busque **Incrustar tipos de interoperabilidad** en la lista de propiedades y cambie su valor a **False**. Del mismo modo, se puede compilar mediante el uso de la opción del compilador -reference en lugar de -link en un símbolo del sistema.

Para agregar formato adicional a la tabla

1. Reemplace las dos llamadas a `AutoFit` en `DisplayInExcel` con la siguiente instrucción.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

El método AutoFormat tiene siete parámetros de valor, todos ellos opcionales. Los argumentos con nombre y los argumentos opcionales permiten proporcionar argumentos para ninguno, algunos o todos ellos. En la instrucción anterior, se proporciona un argumento para uno solo de los parámetros, `Format`. Puesto que `Format` es el primer parámetro de la lista de parámetros, no es necesario proporcionar el nombre de parámetro. Sin embargo, la instrucción sería más fácil de entender si se incluyese el nombre del parámetro, como se muestra en el código siguiente.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. Presione CTRL+F5 para ver el resultado. Otros formatos se indican en la enumeración XlRangeAutoFormat.

3. Compare la instrucción del paso 1 con el siguiente código, que muestra los argumentos necesarios en C# 3.0 y versiones anteriores.

```
// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);
```

Ejemplo

En el código siguiente se muestra el ejemplo completo.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgramminWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a praticular template.
            // Because no argument is sent in this example, Add creates a new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
```

```
            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy method has one
            // optional parameter for specifying a destination. Because no argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to create
            // variables to send in as arguments. You can send the values directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

Vea también

- Type.Missing
- dynamic
- Uso de tipo dinámico
- Argumentos opcionales y con nombre
- Procedimiento para usar argumentos opcionales y con nombre en la programación de Office

Procedimiento Utilizar propiedades indizadas en la programación de interoperabilidad COM (Guía de programación de C#)

18/03/2020 • 4 minutes to read • Edit Online

Las *propiedades indexadas* mejoran la manera en que se usan las propiedades COM con parámetros en la programación de C#. Las propiedades indexadas funcionan junto con otras características de Visual C#, como los argumentos con nombre y opcionales, un nuevo tipo (dinámico) y la información de tipo insertada, para mejorar la programación en Microsoft Office.

En versiones anteriores de C#, los métodos son solo accesibles como propiedades si el método `get` no tiene ningún parámetro y el método `set` tiene solo un parámetro de valor. En cambio, no todas las propiedades COM cumplen esas restricciones. Por ejemplo, la propiedad Range[Object, Object] de Excel tiene un descriptor de acceso `get` que requiere un parámetro para el nombre del intervalo. Antes, como no había acceso directo a la propiedad `Range`, había que usar el método `get_Range` en su lugar, como se muestra en el siguiente ejemplo.

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Las propiedades indexadas, en cambio, permiten escribir lo siguiente:

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

> **NOTE**
>
> En el ejemplo anterior, también se usa la característica argumentos opcionales, que le permite omitir `Type.Missing`.

De manera similar, para establecer el valor de la propiedad `Value` de un objeto Range en C# 3.0 y versiones anteriores, se necesitan dos argumentos. Uno proporciona un argumento para un parámetro opcional que especifica el tipo del valor del intervalo. El otro proporciona el valor de la propiedad `Value`. Estas técnicas se ilustran en los siguientes ejemplos. En ambos ejemplos, se establece el valor de la celda A1 en `Name`.

```
// Visual C# 2008.
targetRange.set_Value(Type.Missing, "Name");
// Or
targetRange.Value2 = "Name";
```

Las propiedades indexadas, en cambio, le permiten escribir el siguiente código.

```
// Visual C# 2010.
targetRange.Value = "Name";
```

No es posible crear propiedades indizadas propias. La característica solo admite el uso de las propiedades indizadas existentes.

Ejemplo

En el código siguiente se muestra un ejemplo completo. Para más información sobre cómo preparar un proyecto con acceso a la API de Office, consulte Procedimiento Tener acceso a objetos de interoperabilidad de Office mediante las características de Visual C# (Guía de programación de C#).

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
            //CSharp2008();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

Vea también

- Argumentos opcionales y con nombre
- dynamic
- Uso de tipo dinámico
- Procedimiento para usar argumentos opcionales y con nombre en la programación de Office
- Procedimiento para acceder a objetos de interoperabilidad de Office mediante características de C#
- Tutorial: Programación de Office

Procedimiento Utilizar la invocación de plataforma para reproducir un archivo de sonido (Guía de programación de C#)

18/03/2020 • 3 minutes to read • Edit Online

En el siguiente ejemplo de código de C# se muestra cómo se usan los servicios de invocación de plataforma para reproducir un archivo de sonido WAV en el sistema operativo Windows.

Ejemplo

En este código de ejemplo se usa DllImportAttribute para importar el punto de entrada del método `winmm.dll` de `PlaySound` como `Form1 PlaySound()`. El ejemplo tiene un formulario Windows Forms simple con un botón. Al hacer clic en el botón, se abre un cuadro de diálogo OpenFileDialog estándar de Windows para que pueda abrir un archivo y reproducirlo. Cuando se selecciona un archivo de sonido, se reproduce mediante el método `PlaySound()` de la biblioteca *winmm.dll*. Para obtener más información sobre este método, vea Using the PlaySound function with Waveform-Audio Files (Uso de la función PlaySound con archivos para forma de onda de sonido). Busque y seleccione un archivo que tenga una extensión .wav y, después, haga clic en **Abrir** para reproducirlo mediante la invocación de plataforma. Un cuadro de texto muestra la ruta de acceso completa del archivo seleccionado.

El cuadro de diálogo **Abrir archivos** se puede filtrar con los siguientes valores para que muestre solo los archivos que tengan la extensión .wav:

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace WinSound
{
    public partial class Form1 : Form
    {
        private TextBox textBox1;
        private Button button1;

        public Form1()  // Constructor.
        {
            InitializeComponent();
        }

        [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true, CharSet = CharSet.Unicode,
ThrowOnUnmappableChar = true)]
        private static extern bool PlaySound(string szSound, System.IntPtr hMod, PlaySoundFlags flags);

        [System.Flags]
        public enum PlaySoundFlags : int
        {
            SND_SYNC = 0x0000,
            SND_ASYNC = 0x0001,
            SND_NODEFAULT = 0x0002,
            SND_LOOP = 0x0008,
            SND_NOSTOP = 0x0010,
            SND_NOWAIT = 0x00002000,
            SND_FILENAME = 0x00020000,
            SND_RESOURCE = 0x00040004
        }

        private void button1_Click(object sender, System.EventArgs e)
        {
            var dialog1 = new OpenFileDialog();

            dialog1.Title = "Browse to find sound file to play";
            dialog1.InitialDirectory = @"c:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            dialog1.FilterIndex = 2;
            dialog1.RestoreDirectory = true;

            if (dialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = dialog1.FileName;
                PlaySound(dialog1.FileName, new System.IntPtr(), PlaySoundFlags.SND_SYNC);
            }
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Including this empty method in the sample because in the IDE,
            // when users click on the form, generates code that looks for a default method
            // with this name. We add it here to prevent confusion for those using the samples.
        }
    }
}
```

Compilación del código

1. Cree un proyecto de aplicación Windows Forms para C# en Visual Studio y asígnele el nombre **WinSound**.

2. Copie el código anterior y péguelo sobre el contenido del archivo *Form1.cs*.

3. Copie el código siguiente y péguelo en el archivo *Form1.Designer.cs*, en el método `InitializeComponent()`,

después de cualquier código existente.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "FIle path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

4. Compile y ejecute el código.

Vea también

- Guía de programación de C#
- Información general sobre interoperabilidad
- Aproximación a la invocación de plataforma
- Serialización de datos con invocación de plataforma

Tutorial: Programación de Office (C# y Visual Basic)

18/03/2020 • 21 minutes to read • Edit Online

Visual Studio presenta características en C# y Visual Basic que mejoran la programación de Microsoft Office. Las características útiles de C# incluyen argumentos opcionales y con nombre, y devuelven valores de tipo `dynamic`. En la programación COM, puede omitir la palabra clave `ref` y obtener acceso a las propiedades indexadas. Las nuevas características de Visual Basic incluyen propiedades implementadas automáticamente, instrucciones de expresiones lambda e inicializadores de colección.

En ambos lenguajes se puede insertar información de tipo, lo que permite la implementación de ensamblados que interactúan con componentes COM sin necesidad de implementar ensamblados de interoperabilidad primarios (PIA) en el equipo del usuario. Para obtener más información, vea Tutorial: Incrustar los tipos de los ensamblados administrados

En este tutorial se muestran estas características en el contexto de la programación de Office, pero muchas de ellas también son útiles en la programación general. En el tutorial, usa una aplicación complemento de Excel para crear un libro de Excel. Después, crea un documento de Word que contiene un vínculo al libro. Por último, ve cómo habilitar y deshabilitar la dependencia de un PIA.

Requisitos previos

Debe tener Microsoft Office Excel y Microsoft Office Word instalados en su equipo para completar este tutorial.

> **NOTE**
> Es posible que el equipo muestre nombres o ubicaciones diferentes para algunos de los elementos de la interfaz de usuario de Visual Studio en las siguientes instrucciones. La edición de Visual Studio que se tenga y la configuración que se utilice determinan estos elementos. Para obtener más información, vea Personalizar el IDE.

Para configurar una aplicación complemento de Excel

1. Inicie Visual Studio.
2. En el menú **Archivo** , elija **Nuevo**y haga clic en **Proyecto**.
3. En el panel **Plantillas instaladas**, expanda **Visual Basic** o **Visual C#** , expanda **Office** y, después, haga clic en el año de la versión del producto de Office.
4. En el panel **Plantillas**, haga clic en **versión de <Excel > Complemento**.
5. En la parte superior del panel **Plantillas**, asegúrese de que **.NET Framework 4** o una versión posterior aparece en el cuadro **Plataforma de destino**.
6. Si quiere, escriba un nombre para el proyecto en el cuadro **Nombre**.
7. Haga clic en **Aceptar**.
8. El nuevo proyecto aparece en el **Explorador de soluciones**.

Para agregar referencias

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el nombre del proyecto y luego haga clic en **Agregar referencia**. Aparecerá el cuadro de diálogo **Agregar referencia**.
2. En la pestaña **Ensamblados**, seleccione **Microsoft.Office.Interop.Excel**, versión `<version>.0.0.0` (para

obtener una clave de los números de versión de productos de Office, vea Versiones de Microsoft), en la lista **Nombre de componente** y, después, mantenga presionada la tecla CTRL y seleccione **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. Si no ve los ensamblados, asegúrese de que están instalados y que se muestran (vea Cómo: Instalar ensamblados de interoperabilidad primarios de Office).

3. Haga clic en **Aceptar**.

Para agregar las instrucciones Imports necesarias o las directivas using

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el archivo **ThisAddIn.vb** o **ThisAddIn.cs** y luego haga clic en **Ver código**.

2. Agregue las siguientes instrucciones `Imports` (Visual Basic) o directivas `using` (C#) en la parte superior del archivo de código si no están presentes.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

Para crear una lista de las cuentas bancarias

1. En el **Explorador de soluciones**, haga clic con el botón derecho en el nombre del proyecto, haga clic en **Agregar** y luego, en **Clase**. Denomine la clase Account.vb si está utilizando Visual Basic o Account.cs si está utilizando C#. Haga clic en **Agregar**.

2. Reemplace la definición de la clase `Account` por el código siguiente. Las definiciones de clase usan *propiedades implementadas automáticamente*. Para obtener más información, vea Auto-Implemented Properties (Propiedades implementadas automáticamente).

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. Para crear una lista `bankAccounts` que contenga dos cuentas, agregue el código siguiente al método `ThisAddIn_Startup` en *ThisAddIn.vb* o *ThisAddIn.cs*. Las declaraciones de lista usan *inicializadores de colección*. Para obtener más información, vea Inicializadores de colección.

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

```
Dim bankAccounts As New List(Of Account) From {
    New Account With {
                          .ID = 345,
                          .Balance = 541.27
                      },
    New Account With {
                          .ID = 123,
                          .Balance = -127.44
                      }
    }
```

Para exportar datos a Excel

1. En el mismo archivo, agregue el siguiente método a la clase `ThisAddIn`. El método configura un libro de Excel, a donde exporta los datos.

```
void DisplayInExcel(IEnumerable<Account> accounts,
           Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

```
Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
               ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub
```

En este método se utilizan dos características de C# nuevas. Ambas características ya existen en Visual Basic.

- El método Add tiene un *parámetro opcional* para especificar una plantilla determinada. Los parámetros opcionales introducidos en C# 4 permiten omitir el argumento para ese parámetro si se desea utilizar el valor predeterminado del parámetro. Dado que en el ejemplo anterior no se envía ningún argumento, `Add` usa la plantilla predeterminada y crea un libro nuevo. La instrucción equivalente en versiones anteriores de C# requiere un argumento de marcador de posición: `excelApp.Workbooks.Add(Type.Missing)`.

  Para obtener más información, vea Argumentos opcionales y con nombre.

- Las propiedades `Range` y `Offset` del objeto Range usan la característica de *propiedades indizadas*. Esta característica permite utilizar estas propiedades de los tipos COM mediante la siguiente sintaxis típica de C#. Las propiedades indizadas también permiten utilizar la propiedad `Value` del objeto `Range`, eliminando la necesidad de utilizar la propiedad `Value2`. La propiedad `Value` está indizada, pero el índice es opcional. Los argumentos opcionales y las propiedades indizadas funcionan conjuntamente en el ejemplo siguiente.

  ```
  // Visual C# 2010 provides indexed properties for COM programming.
  excelApp.Range["A1"].Value = "ID";
  excelApp.ActiveCell.Offset[1, 0].Select();
  ```

  En las versiones anteriores del lenguaje, se requiere la siguiente sintaxis especial.

  ```
  // In Visual C# 2008, you cannot access the Range, Offset, and Value
  // properties directly.
  excelApp.get_Range("A1").Value2 = "ID";
  excelApp.ActiveCell.get_Offset(1, 0).Select();
  ```

  No es posible crear propiedades indizadas propias. La característica solo admite el uso de las propiedades indizadas existentes.

  Para más información, consulte Procedimiento para usar propiedades indizadas en la programación de interoperabilidad COM.

2. Agregue el código siguiente al final de `DisplayInExcel` para ajustar los anchos de columna a fin de

adaptarlos al contenido.

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.
.Columns(1).AutoFit()
.Columns(2).AutoFit()
```

Estas adiciones muestran otra característica de C#: el tratamiento de valores `Object` devueltos por hosts COM, como Office, como si tuvieran un tipo dynamic. Esto sucede automáticamente cuando **Incrustar tipos de interoperabilidad** se establece en su valor predeterminado `True` o, de igual modo, cuando la opción del compilador -link hace referencia al ensamblado. El tipo `dynamic` permite el enlace en tiempo de ejecución, ya disponible en Visual Basic, y evita la conversión explícita que se requiere en C# 3.0 y versiones anteriores del lenguaje.

Por ejemplo, `excelApp.Columns[1]` devuelve `Object` y `AutoFit` es un método Range de Excel. Sin `dynamic`, debe convertir el objeto devuelto por `excelApp.Columns[1]` como una instancia de `Range` antes de llamar al método `AutoFit`.

```
// Casting is required in Visual C# 2008.
((Excel.Range)excelApp.Columns[1]).AutoFit();

// Casting is not required in Visual C# 2010.
excelApp.Columns[1].AutoFit();
```

Para obtener más información sobre cómo insertar tipos de interoperabilidad, consulte los procedimientos "Para buscar la referencia a un PIA" y "Para restaurar la dependencia de un PIA" más adelante en este tema. Para obtener más información sobre `dynamic`, vea dynamic o Uso de tipo dinámico.

Para invocar DisplayInExcel

1. Agregue el código siguiente al final del método `ThisAddIn_StartUp` . La llamada a `DisplayInExcel` contiene dos argumentos. El primer argumento es el nombre de la lista de cuentas que se va a procesar. El segundo argumento es una expresión lambda de varias líneas que define cómo se procesarán los datos. Los valores `ID` y `balance` de cada cuenta se muestran en las celdas adyacentes y la fila se muestra en rojo si el saldo es inferior a cero. Para obtener más información, vea Expresiones lambda.

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

```
DisplayInExcel(bankAccounts,
       Sub(account, cell)
           ' This multiline lambda expression sets custom
           ' processing rules for the bankAccounts.
           cell.Value = account.ID
           cell.Offset(0, 1).Value = account.Balance

           If account.Balance < 0 Then
               cell.Interior.Color = RGB(255, 0, 0)
               cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
           End If
       End Sub)
```

2. Presione F5 para ejecutar el programa. Aparece una hoja de cálculo de Excel que contiene los datos de las cuentas.

Para agregar un documento de Word

1. Agregue el código siguiente al final del método `ThisAddIn_StartUp` para crear un documento de Word que contenga un vínculo al libro de Excel.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

```
Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)
```

Este código muestra algunas de las nuevas características de C#: capacidad para omitir la palabra clave `ref` en programación COM, argumentos con nombre y argumentos opcionales. Estas características ya existen en Visual Basic. El método PasteSpecial tiene siete parámetros, y todos se definen como parámetros de referencia opcionales. Los argumentos opcionales y con nombre permiten designar los parámetros a los que se quiere tener acceso por nombre, y enviar argumentos únicamente a esos parámetros. En este ejemplo se envían argumentos para indicar que se debe crear un vínculo al libro en el Portapapeles (parámetro `Link`), y que el vínculo se mostrará en el documento de Word como un icono (parámetro `DisplayAsIcon`). Visual C# también le permite omitir la palabra clave `ref` para estos argumentos.

Para ejecutar la aplicación

1. Presione F5 para ejecutar la aplicación. Excel se abre y muestra una tabla que contiene la información de las dos cuentas de `bankAccounts`. Entonces aparece un documento de Word que contiene un vínculo a la tabla de Excel.

Para limpiar el proyecto completado

1. En Visual Studio, haga clic en **Limpiar solución** en el menú **Compilación**. De lo contrario, el complemento se ejecutará cada vez que abra Excel en el equipo.

Para buscar la referencia a un PIA

1. Ejecute de nuevo la aplicación, pero no haga clic en **Limpiar solución**.

2. Seleccione **Iniciar**. Busque **Microsoft Visual Studio <versión>** y abra un símbolo del sistema del desarrollador.

3. Escriba `ildasm` en la ventana Símbolo del sistema para desarrolladores de Visual Studio y, luego, presione ENTRAR. Aparecerá la ventana IL DASM.

4. En el menú **Archivo** de la ventana de IL DASM, seleccione **Archivo** > **Abrir**. Haga doble clic en **Visual Studio <versión>** y, después, haga doble clic en **Proyectos**. Abra la carpeta de su proyecto y, en la carpeta bin/Debug, busque *su_proyecto*.dll. Haga doble clic en *su_proyecto*.dll. Una nueva ventana muestra los atributos del proyecto, además de las referencias a otros módulos y ensamblados. Tenga en cuenta que los espacios de nombres `Microsoft.Office.Interop.Excel` y `Microsoft.Office.Interop.Word` se incluyen en el ensamblado. De manera predeterminada en Visual Studio, el compilador importa los tipos necesarios desde un PIA con referencia a su ensamblado.

   Para obtener más información, vea Cómo: Ver el contenido de un ensamblado.

5. Haga doble clic en el icono **MANIFIESTO**. Aparecerá una ventana con una lista de ensamblados que contienen los elementos a los que hace referencia el proyecto. `Microsoft.Office.Interop.Excel` y `Microsoft.Office.Interop.Word` no están incluidos en la lista. Dado que los tipos que su proyecto necesita se han importado en el ensamblado, las referencias a un PIA no son necesarias. Esto facilita la implementación. Los PIA no tienen que estar presentes en el equipo del usuario y, puesto que una aplicación no requiere la implementación de una versión concreta de un PIA, se pueden diseñar aplicaciones que trabajen con varias versiones de Office, siempre que las API necesarias existan en todas las versiones.

   Dado que la implementación de los PIA ya no es necesaria, puede crear una aplicación en escenarios avanzados que funcione con varias versiones de Office, incluidas versiones anteriores. Sin embargo, esto solo funciona si el código no utiliza ninguna API que no esté disponible en la versión de Office con la que está trabajando. No siempre está claro si una API concreta estaba disponible en una versión anterior y por ese motivo no recomendamos trabajar con versiones anteriores de Office.

   > **NOTE**
   > Office no publicó ensamblados PIA antes de Office 2003. Por lo tanto, la única manera de generar un ensamblado de interoperabilidad para Office 2002 o versiones anteriores es mediante la importación de la referencia COM.

6. Cierre la ventana del manifiesto y la del ensamblado.

Para restaurar la dependencia de un PIA

1. En el **Explorador de soluciones**, haga clic en el botón **Mostrar todos los archivos**. Expanda la carpeta **Referencias** y seleccione **Microsoft.Office.Interop.Excel**. Pulse F4 para abrir la ventana **Propiedades**.

2. En la ventana **Propiedades**, cambie la propiedad **Incrustar tipos de interoperabilidad** de **True** a **False**.

3. Repita los pasos 1 y 2 de este procedimiento para `Microsoft.Office.Interop.Word`.

4. En C#, marque como comentario las dos llamadas a `Autofit` al final del método `DisplayInExcel`.

5. Presione F5 para comprobar que el proyecto sigue ejecutándose correctamente.

6. Repita los pasos 1 a 3 del procedimiento anterior para abrir la ventana de ensamblado. Observe que `Microsoft.Office.Interop.Word` y `Microsoft.Office.Interop.Excel` ya no están en la lista de ensamblados insertados.

7. Haga doble clic en el icono **MANIFIESTO** y desplácese por la lista de ensamblados de referencia. Tanto `Microsoft.Office.Interop.Word` como `Microsoft.Office.Interop.Excel` están en la lista. Dado que la aplicación hace referencia a los PIA de Excel y Word y la propiedad **Incrustar tipos de interoperabilidad** se establece en **False**, ambos ensamblados deben existir en el equipo del usuario final.

8. En Visual Studio, haga clic en **Limpiar solución** en el menú **Compilación** para limpiar el proyecto completado.

Vea también

- Propiedades implementadas automáticamente (Visual Basic)
- Propiedades autoimplementadas (C#)
- Inicializadores de colección
- Inicializadores de objeto y colección
- Parámetros opcionales
- Paso de argumentos por posición o por nombre
- Argumentos opcionales y con nombre
- Enlace en tiempo de compilación y en tiempo de ejecución
- dynamic
- Uso de tipo dinámico
- Expresiones lambda (Visual Basic)
- Expresiones lambda (C#)
- Procedimiento para usar propiedades indizadas en la programación de interoperabilidad COM
- Tutorial: Incrustar información de tipos de los ensamblados de Microsoft Office en Visual Studio
- Tutorial: Incrustación de los tipos de los ensamblados administrados
- Tutorial: Creación del primer complemento VSTO para Excel
- Interoperabilidad COM
- Interoperabilidad

Clases COM de ejemplo (Guía de programación de C#)

29/06/2020 • 3 minutes to read • Edit Online

El siguiente es un ejemplo de una clase que se expondría como un objeto COM. Una vez insertado este código de ejemplo en un archivo .cs y agregado al proyecto, establezca la propiedad **Registrar para interoperabilidad COM** en **True**. Para obtener más información, vea Cómo: Registrar un componente para interoperabilidad COM.

Exponer objetos de Visual C# para COM requiere declarar una interfaz de clase, una interfaz de eventos si es necesario y la propia clase. Los miembros de clase deben seguir estas reglas para que sean visibles en COM:

- La clase debe ser pública.
- Las propiedades, métodos y eventos deben ser públicos.
- Las propiedades y métodos deben declararse en la interfaz de clase.
- Los eventos deben declararse en la interfaz de eventos.

Los demás miembros públicos de la clase que no se declaren en estas interfaces no serán visibles para COM, pero lo serán para el resto de objetos de .NET.

Para exponer propiedades y métodos en COM, se deben declarar en la interfaz de clase y marcar con el atributo `DispId`, e implementarlos en la clase. El orden en que se declaran los miembros en la interfaz es el orden usado para la tabla virtual de COM.

Para exponer los eventos de la clase, se deben declarar en la interfaz de eventos y marcarlos con un atributo `DispId`. La clase no debe implementar esta interfaz.

La clase implementa la interfaz de clase y puede implementar más de una interfaz, pero la primera implementación debe ser la interfaz de clase predeterminada. Implemente los métodos y propiedades expuestos para COM aquí. Deben marcarse como públicos y coincidir con las declaraciones de la interfaz de clase. Asimismo, declare los eventos iniciados por la clase aquí. Deben marcarse como públicos y coincidir con las declaraciones de la interfaz de eventos.

Ejemplo

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

Vea también

- Guía de programación de C#
- Interoperabilidad
- Página Compilar (Diseñador de proyectos) (C#)

Referencia de C#

18/03/2020 • 6 minutes to read • Edit Online

En esta sección se proporciona material de referencia sobre palabras clave, operadores, caracteres especiales, directivas de preprocesador, opciones del compilador y errores y advertencias del compilador de C#.

En esta sección

Palabras clave de C#
Ofrece vínculos para información sobre palabras clave de C# y sintaxis.

Operadores de C#
Ofrece vínculos para información sobre operadores de C# y sintaxis.

Caracteres especiales de C#
Proporciona vínculos a información sobre los caracteres especiales contextuales en C# y su uso.

Directivas de preprocesador de C#
Ofrece vínculos para información sobre los comandos del compilador para incrustar en código fuente de C#.

Opciones del compilador de C#
Incluye información sobre las opciones del compilador y cómo usarlas.

Errores del compilador de C#
Incluye fragmentos de código que muestran la causa y la corrección de errores y advertencias del compilador de C#.

Especificación del lenguaje C#
Especificación del lenguaje C# 6.0 Se trata de un borrador de propuesta para el lenguaje C# 6.0. Este documento se perfeccionará por medio del trabajo con el comité de normas de C# de ECMA. La versión 5.0 se ha publicado en diciembre de 2017 como el documento Norma ECMA-334 estándar, quinta edición.

Las características que se han implementado en versiones de C# posteriores a 6.0 se representan en las propuestas de especificación del lenguaje. Estos documentos describen los deltas de la especificación del lenguaje con el fin de agregar estas nuevas características. Están en formato de propuesta de borrador. Estas especificaciones se perfeccionarán y enviarán al comité de normas de ECMA para su revisión formal e incorporación a una versión futura del estándar de C#.

Propuestas de especificación de C# 7.0
Hay una serie de nuevas características implementadas en C# 7.0. Entre ellas, está la coincidencia de patrones, las funciones locales, las declaraciones de variable out, las expresiones throw, los literales binarios y los separadores de dígitos. Esta carpeta contiene las especificaciones para cada una de esas características.

Propuestas de especificación de C# 7.1
Se han agregado nuevas características en C# 7.1. En primer lugar, puede escribir un método `Main` que devuelva `Task` o `Task<int>`. Esto le permite agregar el modificador `async` a

`Main`. La expresión `default` puede usarse sin tipo en ubicaciones en las que sea posible inferir el tipo. Además, se pueden inferir los nombres de los miembros de las tuplas. Por último, se puede usar la coincidencia de patrones con genéricos.

Propuestas de especificación de C# 7.2

Se han agregado una serie de pequeñas características a C# 7.2. Puede pasar argumentos por referencia de solo lectura mediante la palabra clave `in`. Se han realizado diversos cambios de bajo nivel para admitir la seguridad en tiempo de compilación para `Span` y tipos relacionados. Puede usar argumentos con nombre donde los argumentos posteriores son posicionales, en algunos casos. El modificador de acceso `private protected` permite especificar que los llamadores se limitan a los tipos derivados que se implementan en el mismo ensamblado. El operador `?:` se puede resolver en una referencia a una variable. También puede dar formato a números hexadecimales y binarios con un separador de dígito inicial.

Propuestas de especificación de C# 7.3

C# 7.3 es otra versión secundaria que incluye una serie de pequeñas actualizaciones. Puede usar nuevas restricciones en parámetros de tipo genérico. Otros cambios hacen que sea más fácil trabajar con campos `fixed`, incluido el uso de asignaciones `stackalloc`. Las variables locales declaradas con la palabra clave `ref` pueden reasignarse para que hagan referencia a un almacenamiento nuevo. Puede colocar atributos en propiedades implementadas automáticamente que tengan como destino el campo de respaldo generado por el compilador. Se pueden usar variables de expresión en inicializadores. Las tuplas pueden compararse para comprobar si son iguales (o si no lo son). Además, se han realizado algunas mejoras en la resolución de sobrecarga.

Propuestas de especificación de C# 8.0

C# 8.0 está disponible con .NET Core 3.0. Entre las características se incluyen tipos de referencia que aceptan valores NULL, coincidencia de patrones recursiva, métodos de interfaz predeterminados, secuencias asincrónicas, intervalos e índices, using basado en patrones y declaraciones using, asignación de uso combinado de NULL y miembros de instancia de solo lectura.

Secciones relacionadas

Usar el entorno de desarrollo de Visual C#

Ofrece vínculos para los temas de tareas y conceptos y temas que describen el IDE y el Editor.

Guía de programación de C#

Incluye información sobre cómo usar el lenguaje de programación de C#.

Control de versiones del lenguaje C#

29/06/2020 • 8 minutes to read • Edit Online

El compilador de C# más actualizado determina una versión de lenguaje predeterminada basada en los marcos o las plataformas de destino del proyecto. Visual Studio no proporciona una interfaz de usuario para cambiar el valor, pero se puede cambiar si se modifica el archivo *csproj*. La opción predeterminada garantiza que se use la versión del lenguaje más reciente compatible con el marco de trabajo de destino. Se beneficia del acceso a las características de lenguaje más recientes compatibles con el destino del proyecto. Esta opción predeterminada también garantiza que no se use un lenguaje que requiera tipos o el comportamiento en tiempo de ejecución no esté disponible en la plataforma de destino. La elección de una versión del lenguaje más reciente que la predeterminada puede provocar errores en tiempo de compilación y en tiempo de ejecución difíciles de diagnosticar.

Las reglas de este artículo se aplican al compilador ofrecido con Visual Studio 2019 o al SDK de .NET Core 3.0. Los compiladores de C# que forman parte de la instalación de Visual Studio 2017 o versiones anteriores del SDK de .NET Core tienen como destino C# 7.0 de forma predeterminada.

C# 8.0 (y versiones posteriores) solo se admite en .NET Core 3.x y versiones más recientes. Muchas de las características más recientes requieren características de biblioteca y runtime introducidas en .NET Core 3.x:

- La implementación de interfaz predeterminada requiere nuevas características en el CLR de .NET Core 3.0.
- Las secuencias asincrónicas requieren los nuevos tipos System.IAsyncDisposable, System.Collections.Generic.IAsyncEnumerable<T> y System.Collections.Generic.IAsyncEnumerator<T>.
- Los índices y los intervalos requieren los nuevos tipos System.Indexy System.Range.
- Los tipos de referencia que admiten un valor NULL hacen uso de varios atributos para proporcionar mejores advertencias. Esos atributos se han agregado en .NET Core 3.0. Otras plataformas de destino no se han anotado con ninguno de estos atributos. Esto significa que es posible que las advertencias que admiten un valor NULL no reflejen con precisión los posibles problemas.

Valores predeterminados

El compilador determina un valor predeterminado según estas reglas:

MARCO DE DESTINO	VERSION	VERSIÓN PREDETERMINADA DEL LENGUAJE C#
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	todo	C# 7.3

Cuando el proyecto tiene como destino un marco en versión preliminar que tenga una versión de lenguaje preliminar correspondiente, la versión de lenguaje que se usa es la que está en versión preliminar. Puede usar

las características más recientes con esa versión preliminar en cualquier entorno, sin que afecte a los proyectos que tienen como destino una versión de .NET Core publicada.

Invalidación de un valor predeterminado

Si debe especificar su versión de C# explícitamente, puede hacerlo de varias maneras:

- Editar manualmente el archivo del proyecto.
- Establecer la versión del lenguaje para varios proyectos en un subdirectorio.
- Configurar la opción dl compilador Reemplace la opción del compilador `-langversion`.

Edición del archivo del proyecto

Puede establecer la versión del lenguaje en el archivo del proyecto. Por ejemplo, si quiere acceder explícitamente a las características en versión preliminar, agregue un elemento similar al siguiente:

```
<PropertyGroup>
   <LangVersion>preview</LangVersion>
</PropertyGroup>
```

El valor `preview` usa la versión preliminar más reciente disponible del lenguaje C# que admite el compilador.

Configurar varios proyectos

Para configurar varios proyectos, se puede crear un archivo **Directory.Build.props** que contenga el elemento `<LangVersion>`. Por lo general, esto se hace en el directorio de la solución. Agregue lo siguiente a un archivo **Directory.Build.props** en el directorio de la solución:

```
<Project>
 <PropertyGroup>
   <LangVersion>preview</LangVersion>
 </PropertyGroup>
</Project>
```

Las compilaciones de todos los subdirectorios del directorio que contenga ese archivo usarán la sintaxis de la versión preliminar de C#. Para obtener más información, consulte el artículo Personalizar una compilación.

Referencia de la versión del lenguaje C#

En la siguiente tabla se muestran las versiones actuales del lenguaje C#. Es posible que el compilador no entienda necesariamente todos los valores si es más antiguo. Si instala .NET Core 3.0 o posterior, tiene acceso a todo lo que aparece.

VALOR	SIGNIFICADO
`preview`	El compilador acepta toda la sintaxis de lenguaje válida de la última versión preliminar.
`latest`	El compilador acepta la sintaxis de la última versión del compilador (incluida las versión secundaria).
`latestMajor` (`default`)	El compilador acepta la sintaxis de la versión principal más reciente del compilador.
`8.0`	El compilador acepta solo la sintaxis que se incluye en C# 8.0 o versiones anteriores.

VALOR	SIGNIFICADO
`7.3`	El compilador acepta solo la sintaxis que se incluye en C# 7.3 o versiones anteriores.
`7.2`	El compilador acepta solo la sintaxis que se incluye en C# 7.2 o versiones anteriores.
`7.1`	El compilador acepta solo la sintaxis que se incluye en C# 7.1 o versiones anteriores.
`7`	El compilador acepta solo la sintaxis que se incluye en C# 7.0 o versiones anteriores.
`6`	El compilador acepta solo la sintaxis que se incluye en C# 6.0 o versiones anteriores.
`5`	El compilador acepta solo la sintaxis que se incluye en C# 5.0 o versiones anteriores.
`4`	El compilador acepta solo la sintaxis que se incluye en C# 4.0 o versiones anteriores.
`3`	El compilador acepta solo la sintaxis que se incluye en C# 3.0 o versiones anteriores.
`ISO-2` (o `2`)	El compilador acepta solo la sintaxis que se incluye en ISO/IEC 23270:2006 C# (2.0).
`ISO-1` (o `1`)	El compilador acepta solo la sintaxis que se incluye en ISO/IEC 23270:2003 C# (1.0/1.2).

TIP

Abra el Símbolo del sistema para desarrolladores de Visual Studio y ejecute el siguiente comando para ver la lista de versiones de idioma disponibles en la máquina.

```
csc -langversion:?
```

Al cuestionar la opción de compilación -langversion de este modo, se imprimirá algo similar a lo siguiente:

```
Supported language versions:
default
1
2
3
4
5
6
7.0
7.1
7.2
7.3
8.0 (default)
latestmajor
preview
latest
```

Palabras clave de C#

24/04/2020 • 2 minutes to read • Edit Online

Las palabras clave son identificadores reservados predefinidos que tienen un significado especial para el compilador. No podrá utilizarlos como identificadores en el programa a no ser que incluyan `@` como prefijo. Por ejemplo, `@if` es un identificador válido, pero `if` no lo es, porque `if` es una palabra clave.

En la primera tabla de este tema se muestran las palabras clave que son identificadores reservados en cualquier parte de un programa en C#. En la segunda tabla de este tema se enumeran las palabras clave contextuales en C#. Las palabras clave contextuales tienen un significado especial solo en un contexto de programa limitado y pueden utilizarse como identificadores fuera de ese contexto. Por lo general, cuando se agregan nuevas palabras clave al lenguaje C#, se agregan como palabras clave contextuales para evitar la interrupción de los programas escritos en versiones anteriores.

abstract	as	base	bool	
break	byte	case	catch	
char	checked	class	const	
continue	decimal	default	delegate	
do	double	else	enum	
event	explicit	extern	false	
finally	fixed	float	for	
foreach	goto	if	implicit	
in	int	interface	internal	
is	lock	long	namespace	
new	null	object	operator	
out	override	params	private	
protected	public	readonly	ref	
return	sbyte	sealed	short	
sizeof	stackalloc	static	string	
struct	switch	this	throw	
true	try	typeof	uint	

ulong	unchecked	unsafe	ushort	
using	virtual	void	volatile	
while				

Palabras clave contextuales

Las palabras clave contextuales se usan para proporcionar un significado específico en el código, pero no son una palabra reservada en C#. Algunas palabras clave contextuales, como `partial` y `where`, tienen significados especiales en dos o más contextos.

add	alias	ascending
async	await	by
descending	dynamic	equals
from	get	global
group	into	join
let	nameof	on
orderby	partial (tipo)	partial (método)
remove	select	set
unmanaged (restricción de tipo genérico)	value	var
when (condición de filtro)	where (restricción de tipo genérico)	where (cláusula de consulta)
yield		

Vea también

- Referencia de C#

Tipos de valor (Referencia de C#)

18/03/2020 • 5 minutes to read • Edit Online

Los *tipos de valor* y los tipos de referencia son las dos categorías principales de tipos de C#. Una variable de un tipo de valor contiene una instancia del tipo. Esto difiere de una variable de un tipo de referencia, que contiene una referencia a una instancia del tipo. De forma predeterminada, al asignar, pasar un argumento a un método o devolver el resultado de un método, se copian los valores de variable. En el caso de las variables de tipo de valor, se copian las instancias de tipo correspondientes. En el ejemplo siguiente se muestra ese comportamiento:

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified: {p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}
// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

Como se muestra en el ejemplo anterior, las operaciones en una variable de tipo de valor solo afectan a esa instancia del tipo de valor, almacenado en la variable.

Si un tipo de valor contiene un miembro de datos de un tipo de referencia, solo se copia la referencia a la instancia del tipo de referencia al copiar una instancia de tipo de valor. Tanto la instancia de tipo de valor original como la copia tienen acceso a la misma instancia de tipo de referencia. En el ejemplo siguiente se muestra ese comportamiento:

```
using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ", tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1);  // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1);  // output: 0 [A, B]
        Console.WriteLine(n2);  // output: 7 [A, B]
    }
}
```

NOTE

Para que el código sea menos propenso a errores y más sólido, defina y use tipos de valor inmutables. En este artículo se usan tipos de valor mutables solo con fines de demostración.

Clases de tipos de valor

Un tipo de valor puede ser de una de las dos clases siguientes:

- un tipo de estructura, que encapsula los datos y la funcionalidad relacionada;
- un tipo de enumeración, que se define mediante un conjunto de constantes con nombre y representa una opción o una combinación de opciones.

Un tipo de valor que admite valores NULL `T?` representa todos los valores de su tipo de valor subyacente `T` y un valor NULL adicional. No se puede asignar `null` a una variable de un tipo de valor, a menos que sea un tipo de valor que acepte valores NULL.

Tipos de valor integrados

C# proporciona los siguientes tipos de valor integrados, también conocidos como *tipos simples*:

- Tipos numéricos integrales
- Tipos numéricos de punto flotante
- bool, que representa un valor booleano

- char, que representa un carácter Unicode UTF-16

Todos los tipos simples son tipos de estructuras y se diferencian de otros tipos de estructuras en que permiten determinadas operaciones adicionales:

- Se pueden usar literales para proporcionar un valor de un tipo simple. Por ejemplo, `'A'` es un literal del tipo `char` y `2001` es un literal del tipo `int`.
- Puede declarar constantes de los tipos simples con la palabra clave const. No es posible tener constantes de otros tipos de estructuras.
- Las expresiones constantes, cuyos operandos son todas constantes de los tipos simples, se evalúan en tiempo de compilación.

A partir de C# 7.0, C# admite tuplas de valor. Una tupla de valor es un tipo de valor, pero no un tipo simple.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Tipos de valor
- Tipos simples
- Variables

Vea también

- Referencia de C#
- System.ValueType
- Tipos de referencia

Tipos numéricos enteros (referencia de C#)

04/05/2020 • 6 minutes to read • Edit Online

Los *tipos numéricos integrales* representan números enteros. Todos los tipos numéricos integrales son tipos de valor. También son tipos simples y se pueden inicializar con literales. Todos los tipos numéricos enteros admiten operadores aritméticos, lógicos bit a bit, de comparación y de igualdad.

Características de los tipos enteros

C# admite los siguientes tipos enteros predefinidos:

PALABRA CLAVE/TIPO DE C#	INTERVALO	TAMAÑO	TIPO DE .NET
`sbyte`	De -128 a 127	Entero de 8 bits con signo	System.SByte
`byte`	De 0 a 255	Entero de 8 bits sin signo	System.Byte
`short`	De -32 768 a 32 767	Entero de 16 bits con signo	System.Int16
`ushort`	De 0 a 65.535	Entero de 16 bits sin signo	System.UInt16
`int`	De -2.147.483.648 a 2.147.483.647	Entero de 32 bits con signo	System.Int32
`uint`	De 0 a 4.294.967.295	Entero de 32 bits sin signo	System.UInt32
`long`	De -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	Entero de 64 bits con signo	System.Int64
`ulong`	De 0 a 18.446.744.073.709.551.615	Entero de 64 bits sin signo	System.UInt64

En la tabla anterior, cada palabra clave de tipo de C# de la columna ubicada más a la izquierda es un alias del tipo de .NET correspondiente. Son intercambiables. Por ejemplo, en las declaraciones siguientes se declaran variables del mismo tipo:

```
int a = 123;
System.Int32 b = 123;
```

El valor predeterminado de cada tipo entero es cero, `0`. Cada uno de los tipos enteros tiene las constantes `MinValue` y `MaxValue` que proporcionan el valor mínimo y máximo de ese tipo.

Use la estructura System.Numerics.BigInteger para representar un entero con signo sin límite superior ni inferior.

Literales enteros

Los literales enteros pueden ser

- *decimales*: sin ningún prefijo
- *hexadecimales*: con el prefijo de `0x` o `0X`
- *binarios*: con el prefijo `0b` o `0B` (disponible en C# 7.0 y versiones posteriores)

En el código siguiente se muestra un ejemplo de cada uno de ellos:

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

En el ejemplo anterior también se muestra el uso de `_` como un *separador de dígitos*, que se admite a partir de C# 7.0. Puede usar el separador de dígitos con todos los tipos de literales numéricos.

El tipo de un literal entero viene determinado por su sufijo, como se indica a continuación:

- Si el literal no tiene sufijo, su tipo es el primero de los siguientes tipos en el que se puede representar su valor: `int`, `uint`, `long`, `ulong`.
- Si un literal entero tiene el sufijo `U` o `u`, su tipo es el primero de los siguientes tipos en el que se puede representar su valor: `uint`, `ulong`.
- Si un literal entero tiene el sufijo `L` o `l`, su tipo es el primero de los siguientes tipos en el que se puede representar su valor: `long`, `ulong`.

  > **NOTE**
  > Puede usar la letra minúscula `l` como sufijo. Sin embargo, esto genera una advertencia del compilador porque la letra `l` se confunde fácilmente con el dígito `1`. Use `L` para mayor claridad.

- Si el literal tiene como sufijo `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU` o `lu`, su tipo es `ulong`.

Si el valor que representa un literal entero supera UInt64.MaxValue, se produce un error de compilación CS1021.

Si el tipo determinado de un literal entero es `int` y el valor que representa el literal está dentro del rango del tipo de destino, el valor se puede convertir de forma implícita en `sbyte`, `byte`, `short`, `ushort`, `uint` o `ulong`:

```
byte a = 17;
byte b = 300;   // CS0031: Constant value '300' cannot be converted to a 'byte'
```

Como se muestra en el ejemplo anterior, si el valor del literal no está dentro del intervalo del tipo de destino, se produce el error CS0031 del compilador.

También puede usar una conversión para convertir el valor representado por un literal entero en un tipo que no sea el determinado del literal:

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

Conversiones

Puede convertir un tipo numérico entero en cualquier otro tipo numérico entero. Si el tipo de destino puede almacenar todos los valores del tipo de origen, la conversión es implícita. De lo contrario, debe usar una expresión Cast para realizar una conversión explícita. Para obtener más información, consulte Conversiones numéricas integradas.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Tipos enteros
- Literales enteros

Vea también

- Referencia de C#
- Tipos de valor
- Tipos de punto flotante
- Cadenas con formato numérico estándar
- Valores numéricos en .NET

Tipos numéricos de punto flotante (referencia de C#)

29/06/2020 • 6 minutes to read • Edit Online

Los *tipos numéricos de punto flotante* representan números reales. Todos los tipos numéricos de punto flotante son tipos de valor. También son tipos simples y se pueden inicializar con literales. Todos los tipos de punto flotante numéricos admiten operadores aritméticos, de comparación y de igualdad.

Características de los tipos de punto flotante

C# admite los siguientes tipos de punto flotante predefinidos:

PALABRA CLAVE/TIPO DE C#	INTERVALO APROXIMADO	PRECISIÓN	TAMAÑO	TIPO DE .NET
`float`	De $\pm 1{,}5 \times 10^{-45}$ a $\pm 3{,}4 \times 10^{38}$	De 6 a 9 dígitos aproximadamente	4 bytes	System.Single
`double`	De $\pm 5{,}0 \times 10^{-324}$ a $\pm 1{,}7 \times 10^{308}$	De 15 a 17 dígitos aproximadamente	8 bytes	System.Double
`decimal`	De $\pm 1{,}0 \times 10^{-28}$ to $\pm 7{,}9228 \times 10^{28}$	28-29 dígitos	16 bytes	System.Decimal

En la tabla anterior, cada palabra clave de tipo de C# de la columna ubicada más a la izquierda es un alias del tipo de .NET correspondiente. Son intercambiables. Por ejemplo, en las declaraciones siguientes se declaran variables del mismo tipo:

```
double a = 12.3;
System.Double b = 12.3;
```

El valor predeterminado de cada tipo de punto flotante es cero, `0`. Cada uno de los tipos de punto flotante tiene las constantes `MinValue` y `MaxValue` que proporcionan el valor finito mínimo y máximo de ese tipo. Los tipos `float` y `double` también brindan constantes que representan valores infinitos y no numéricos. Por ejemplo, el tipo `double` ofrece las siguientes constantes: Double.NaN, Double.NegativeInfinity y Double.PositiveInfinity.

Como el tipo `decimal` tiene más precisión y un intervalo más reducido que `float` y `double`, es adecuado para los cálculos financieros y monetarios.

En una expresión, puede combinar tipos enteros y tipos `float` y `double`. En este caso, los tipos enteros se convierten implícitamente en uno de los tipos de punto flotante y, si es necesario, el tipo `float` se convierte implícitamente en `double`. La expresión se evalúa de la siguiente forma:

- Si hay un tipo `double` en la expresión, esta se evalúa como `double`, o bien como `bool` en comparaciones relacionales o de igualdad.
- Si no hay un tipo `double` en la expresión, esta se evalúa como `float`, o bien como `bool` en comparaciones relacionales o de igualdad.

También es posible combinar en una expresión tipos enteros y el tipo `decimal`. En este caso, los tipos

enteros se convierten implícitamente en el tipo `decimal` y la expresión se evalúa como `decimal`, o bien como `bool` en comparaciones relacionales y de igualdad.

En una expresión, no se puede mezclar el tipo `decimal` con los tipos `float` y `double`. En este caso, si quiere realizar operaciones aritméticas, de comparación o de igualdad, debe convertir explícitamente los operandos del tipo `decimal` o a este mismo tipo, como se muestra en el ejemplo siguiente:

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

Puede usar cadenas con formato numérico estándar o cadenas con formato numérico personalizado para dar formato a un valor de punto flotante.

Literales reales

El tipo de un literal real viene determinado por su sufijo, como se indica a continuación:

- El literal sin sufijo o con el sufijo `d` o `D` es del tipo `double`
- El literal con el sufijo `f` o `F` es del tipo `float`
- El literal con el sufijo `m` o `M` es del tipo `decimal`

En el código siguiente se muestra un ejemplo de cada uno de ellos:

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

En el ejemplo anterior también se muestra el uso de `_` como un *separador de dígitos*, que se admite a partir de C# 7.0. Puede usar el separador de dígitos con todos los tipos de literales numéricos.

También puede usar la notación científica; es decir, especificar una parte exponencial de un literal real, como se muestra en el ejemplo siguiente:

```
double d = 0.42e2;
Console.WriteLine(d);  // output 42

float f = 134.45E-2f;
Console.WriteLine(f);  // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m);  // output: 1500000
```

Conversiones

Solo hay una conversión implícita entre los tipos numéricos de punto flotante: de `float` a `double`. Sin embargo, puede convertir un tipo de punto flotante a cualquier otro tipo de punto flotante con la conversión explícita. Para obtener más información, consulte Conversiones numéricas integradas.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Tipos de punto flotante
- El tipo decimal
- Literales reales

Vea también

- Referencia de C#
- Tipos de valor
- Tipos enteros
- Cadenas con formato numérico estándar
- Valores numéricos en .NET
- System.Numerics.Complex

Conversiones numéricas integradas (referencia de C#)

16/04/2020 • 8 minutes to read • Edit Online

C# proporciona un conjunto de tipos numéricos enteros y de punto flotante. Existe una conversión entre dos tipos numéricos, ya sea implícita o explícita. Debe usar una expresión Cast para realizar una conversión explícita.

Conversiones numéricas implícitas

En la tabla siguiente se muestran las conversiones implícitas predefinidas entre los tipos numéricos integrados:

DE	EN
sbyte	`short`, `int`, `long`, `float`, `double` o `decimal`
byte	`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` o `decimal`
short	`int`, `long`, `float`, `double` o `decimal`
ushort	`int`, `uint`, `long`, `ulong`, `float`, `double` o `decimal`
int	`long`, `float`, `double` o `decimal`
uint	`long`, `ulong`, `float`, `double` o `decimal`
long	`float`, `double` o `decimal`
ulong	`float`, `double` o `decimal`
float	`double`

NOTE

Las conversiones implícitas de `int`, `uint`, `long` o `ulong` a `float` y de `long` o `ulong` a `double` pueden provocar una pérdida de precisión, pero nunca una pérdida de un orden de magnitud. El resto de conversiones numéricas implícitas nunca pierden información.

Tenga en cuenta también lo siguiente:

- Cualquier tipo numérico entero es implícitamente convertible en cualquier tipo numérico de punto flotante.
- No hay ninguna conversión implícita a los tipos `byte` y `sbyte`. No hay ninguna conversión implícita de los tipos `double` y `decimal`.
- No hay ninguna conversión implícita entre el tipo `decimal` y el tipo `float` o `double`.

- Un valor de una expresión constante de tipo `int` (por ejemplo, un valor representado por un literal entero) se puede convertir implícitamente en `sbyte`, `byte`, `short`, `ushort`, `uint` o `ulong`, siempre que esté dentro del rango del tipo de destino:

  ```
  byte a = 13;
  byte b = 300;  // CS0031: Constant value '300' cannot be converted to a 'byte'
  ```

  Como se muestra en el ejemplo anterior, si el valor constante no está dentro del rango del tipo de destino, se produce el error del compilador CS0031.

Conversiones numéricas explícitas

La siguiente tabla muestra las conversiones explícitas predefinidas entre tipos numéricos integrados para los que no hay ninguna conversión implícita:

DE	EN
sbyte	`byte`, `ushort`, `uint` o `ulong`
byte	`sbyte`
short	`sbyte`, `byte`, `ushort`, `uint` o `ulong`
ushort	`sbyte`, `byte` o `short`
int	`sbyte`, `byte`, `short`, `ushort`, `uint` o `ulong`
uint	`sbyte`, `byte`, `short`, `ushort` o `int`
long	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint` o `ulong`
ulong	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint` o `long`
float	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` o `decimal`
double	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float` o `decimal`
decimal	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float` o `double`

> **NOTE**
> Una conversión numérica explícita podría provocar la pérdida de datos o producir una excepción, normalmente una de tipo OverflowException.

Tenga en cuenta también lo siguiente:

- Al convertir un valor de tipo entero en otro tipo entero, el resultado depende del contexto de comprobación de desbordamiento. En un contexto comprobado, la conversión se realiza correctamente si el valor de origen está dentro del intervalo del tipo de destino. De lo contrario, se produce una

excepción OverflowException. En un contexto no comprobado, la conversión siempre se realiza correctamente y continúa así:

  - Si el tipo de origen es mayor que el tipo de destino, el valor de origen se trunca al descartar sus bits "extra" más significativos. El resultado se trata como un valor del tipo de destino.
  - Si el tipo de origen es menor que el tipo de destino, el valor de origen se amplía mediante signos o ceros para que tenga el mismo tamaño que el tipo de destino. La ampliación mediante signos se usa si el tipo de origen tiene signo; se emplea ampliación mediante ceros si el tipo de origen no tiene signo. El resultado se trata como un valor del tipo de destino.
  - Si el tipo de origen es del mismo tamaño que el tipo de destino, el valor de origen se trata como un valor del tipo de destino.

- Cuando convierte un valor `decimal` en un tipo entero, este valor se redondea hacia cero al valor entero más cercano. Si el valor entero resultante está fuera del rango del tipo de destino, se genera una excepción OverflowException.
- Al convertir un valor `double` o `float` en un tipo entero, este valor se redondea hacia cero al valor entero más cercano. Si el valor entero resultante está fuera del intervalo del tipo de destino, el resultado depende del contexto de comprobación de desbordamiento. En un contexto comprobado, se genera una excepción OverflowException, mientras que en un contexto no comprobado, el resultado es un valor no especificado del tipo de destino.
- Cuando convierte `double` en `float`, el valor `double` se redondea al valor `float` más cercano. Si el valor `double` es demasiado pequeño o demasiado grande para adaptarse al tipo `float`, el resultado será cero o infinito.
- Cuando convierte `float` o `double` en `decimal`, el valor de origen se convierte en la representación `decimal` y se redondea al número más cercano después de la posición decimal 28 si es necesario. Dependiendo del valor que tenga el valor de origen, puede producirse uno de los siguientes resultados:
  - Si el valor de origen es demasiado pequeño para representarse como `decimal`, el resultado se convierte en cero.
  - Si el valor de origen es NaN (no es un número), infinito o demasiado grande para representarse como un `decimal`, se genera OverflowException.
- Cuando se convierte `decimal` en `float` o `double`, el valor de origen se redondea al valor `float` o `double` más cercano, respectivamente.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Conversiones numéricas implícitas
- Conversiones numéricas explícitas

Vea también

- Referencia de C#
- Conversiones de tipos

bool (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave de tipo `bool` es un alias para el tipo de estructura de .NET System.Boolean que representa un valor booleano que puede ser `true` o `false`.

Para realizar operaciones lógicas con valores del tipo `bool`, use operadores lógicos booleanos. El tipo `bool` es el tipo de resultado de los operadores de comparación e igualdad. Una expresión `bool` puede ser una expresión condicional de control en las instrucciones if, do, while y for, así como en el operador condicional `?:`.

El valor predeterminado del tipo `bool` es `false`.

Literales

Puede usar los literales `true` y `false` para inicializar una variable `bool` o para pasar un valor `bool`:

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not checked
```

Lógica booleana de tres valores

Use el tipo `bool?` que acepta valores NULL si tiene que admitir la lógica de tres valores (por ejemplo, si trabaja con bases de datos que admiten un tipo booleano de tres valores). En el caso de los operandos `bool?`, los operadores predefinidos `&` y `|` admiten la lógica de tres valores. Para más información, consulte la sección Operadores lógicos booleanos que aceptan valores NULL del artículo Operadores lógicos booleanos.

Para más información sobre los tipos de valor que admiten un valor NULL, consulte Tipos de valor que admiten un valor NULL.

Conversiones

C# solo proporciona dos conversiones que implican al tipo `bool`. Son una conversión implícita al tipo `bool?` que acepta valores NULL correspondiente y una conversión explícita del tipo `bool?`. Sin embargo, .NET proporciona métodos adicionales que se pueden usar para realizar una conversión al tipo `bool`, o bien revertirla. Para obtener más información, vea la sección Convertir a y desde valores booleanos de la página de referencia de la API System.Boolean.

especificación del lenguaje C#

Para obtener más información, vea la sección Tipo bool de la especificación del lenguaje C#.

Vea también

- Referencia de C#
- Tipos de valor
- Operadores true y false

char (referencia de C#)

29/06/2020 • 3 minutes to read • Edit Online

La palabra clave de tipo `char` es un alias para el tipo de estructura de .NET System.Char que representa un carácter Unicode UTF-16.

TIPO	INTERVALO	TAMAÑO	TIPO DE .NET
`char`	U+0000 a U+FFFF	16 bits	System.Char

El valor predeterminado del tipo `char` es `\0`, es decir, U+0000.

El tipo `char` admite operadores de comparación, igualdad, incremento y decremento. Además, en el caso de los operandos `char`, los operadores aritméticos y lógicos bit a bit realizan una operación en los códigos de caracteres correspondientes y producen el resultado del tipo `int`.

El tipo string representa el texto como una secuencia de valores `char`.

Literales

Puede especificar un valor de `char` con:

- un literal de carácter.
- una secuencia de escape Unicode, que es `\u` seguido de la representación hexadecimal de cuatro símbolos de un código de carácter.
- una secuencia de escape hexadecimal, que es `\x` seguido de la representación hexadecimal de un código de carácter.

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars));  // output: j j j j
```

Como se muestra en el ejemplo anterior, también puede convertir el valor de un código de carácter en el valor de `char` correspondiente.

> **NOTE**
>
> En el caso de una secuencia de escape Unicode, debe especificar los cuatro dígitos hexadecimales. Es decir, `\u006A` es una secuencia de escape válida, mientras que `\u06A` y `\u6A` no son válidas.
>
> En el caso de una secuencia de escape hexadecimal, puede omitir los ceros a la izquierda. Es decir, las secuencias de escape `\x006A`, `\x06A` y `\x6A` son válidas y se corresponden con el mismo carácter.

Conversiones

El tipo `char` se puede convertir implícitamente en los tipos enteros siguientes: `ushort`, `int`, `uint`, `long` y

`ulong`. También se puede convertir implícitamente en los tipos numéricos de punto flotante integrados: `float`, `double` y `decimal`. Se puede convertir explícitamente en los tipos enteros `sbyte`, `byte` y `short`.

No hay ninguna conversión implícita de otros tipos al tipo `char`. Sin embargo, cualquier tipo numérico entero o de punto flotante es implícitamente convertible a `char`.

Especificación del lenguaje C#

Para obtener más información, consulte la sección Tipos enteros de Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Tipos de valor
- Cadenas
- System.Text.Rune
- Codificación de caracteres de .NET

Tipos de enumeración (referencia de C#)

04/05/2020 • 6 minutes to read • Edit Online

Un *tipo de enumeración* es un tipo de valor definido por un conjunto de constantes con nombre del tipo numérico integral subyacente. Para definir un tipo de enumeración, use la palabra clave `enum` y especifique los nombres de *miembros de enumeración*:

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

De forma predeterminada, los valores de constante asociados de miembros de enumeración son del tipo `int`; comienzan con cero y aumentan en uno después del orden del texto de la definición. Puede especificar explícitamente cualquier otro tipo de numérico entero como un tipo subyacente de un tipo de enumeración. También puede especificar explícitamente los valores de constante asociados, como se muestra en el ejemplo siguiente:

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

No se puede definir un método dentro de la definición de un tipo de enumeración. Para agregar funcionalidad a un tipo de enumeración, cree un método de extensión.

El valor predeterminado de un tipo de enumeración `E` es el valor generado por la expresión `(E)0`, incluso si cero no tiene el miembro de enumeración correspondiente.

Un tipo de enumeración se usa para representar una opción de un conjunto de valores mutuamente excluyentes o una combinación de opciones. Para representar una combinación de opciones, defina un tipo de enumeración como marcas de bits.

Tipos de enumeración como marcas de bits

Si quiere que un tipo de enumeración represente una combinación de opciones, defina los miembros de enumeración de esas opciones de modo que una opción individual sea un campo de bits. Es decir, los valores asociados de esos miembros de enumeración deben ser las potencias de dos. Luego, puede usar los operadores lógicos bit a bit `|` o `&` para combinar o formar intersección de combinaciones de opciones, respectivamente. Para indicar que un tipo de enumeración declara campos de bits, aplíquele el atributo Flags. Como se muestra en el ejemplo siguiente, también puede incluir algunas combinaciones típicas en la definición de un tipo de enumeración.

```
[Flags]
public enum Days
{
    None      = 0b_0000_0000,  // 0
    Monday    = 0b_0000_0001,  // 1
    Tuesday   = 0b_0000_0010,  // 2
    Wednesday = 0b_0000_0100,  // 4
    Thursday  = 0b_0000_1000,  // 8
    Friday    = 0b_0001_0000,  // 16
    Saturday  = 0b_0010_0000,  // 32
    Sunday    = 0b_0100_0000,  // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays & workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) == Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday: {isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}
```

Para más información y ver algunos ejemplos, consulte la página de referencia de API de System.FlagsAttribute y la sección miembros no exclusivos y el atributo Flags de la página de referencia de API de System.Enum.

El tipo System.Enum y la restricción de enumeración

El tipo System.Enum es la clase base abstracta de todos los tipos de enumeración. Proporciona una serie de métodos para obtener información sobre un tipo de enumeración y sus valores. Para más información y ver algunos ejemplos, consulte la página de referencia de la API System.Enum.

A partir C# 7.3, puede usar `System.Enum` en una restricción de clase base (conocida como la restricción de enumeración) para especificar que un parámetro de tipo es un tipo de enumeración.

Conversiones

Para cualquier tipo de enumeración, existen conversiones explícitas entre el tipo de enumeración y su tipo entero subyacente. Si convierte (usacast) un valor de enumeración a su tipo subyacente, el resultado es el valor entero asociado de un miembro de enumeración.

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output: Integral value of
Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

Use el método Enum.IsDefined para determinar si un tipo de enumeración contiene un miembro de enumeración con el valor asociado determinado.

Para cualquier tipo de enumeración, existen conversiones boxing y unboxing a y desde el tipo System.Enum, respectivamente.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Enumeraciones
- Operaciones y valores de enumeración
- Operadores lógicos de enumeración
- Operadores de comparación de enumeración
- Conversiones de enumeración explícitas
- Conversiones de enumeración implícitas

Vea también

- Referencia de C#
- Cadenas de formato de enumeración
- Instrucciones de diseño: diseño de enumeraciones
- Instrucciones de diseño: convenciones de nomenclatura de enumeración
- switch (Instrucción)

Tipos de estructura (Referencia de C#)

04/05/2020 • 14 minutes to read • Edit Online

Un *tipo de estructura* (o *tipo struct*) es un tipo de valor que puede encapsular datos y funcionalidad relacionada. Para definir un tipo de estructura se usa la palabra clave `struct`:

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Los tipos de estructura tienen *semántica de valores*. Es decir, una variable de un tipo de estructura contiene una instancia del tipo. De forma predeterminada, los valores de variable se copian al asignar, pasar un argumento a un método o devolver el resultado de un método. En el caso de una variable de tipo de estructura, se copia una instancia del tipo. Para más información, vea Tipos de valor.

Normalmente, los tipos de estructura se usan para diseñar tipos de pequeño tamaño centrados en datos que proporcionan poco o ningún comportamiento. Por ejemplo, en .NET se usan los tipos de estructura para representar un número (entero y real), un valor booleano, un caracter Unicode, una instancia de tiempo. Si le interesa el comportamiento de un tipo, considere la posibilidad de definir una clase. Los tipos de clase tienen *semántica de referencias*. Es decir, una variable de un tipo de clase contiene una referencia a una instancia del tipo, no la propia instancia.

Dado que los tipos de estructura tienen semántica del valor, le recomendamos que defina tipos de estructura *inmutables*.

Estructura `readonly`

A partir de C# 7.2, usa el modificador `readonly` para declarar que un tipo de estructura es inmutable:

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

Todos los miembros de datos de una estructura `readonly` debe ser de solo lectura tal como se indica a continuación:

- Cualquier declaración de campo debe tener el modificador `readonly`
- Cualquier propiedad, incluidas las implementadas automáticamente, deben ser de solo lectura

Esto garantiza que ningún miembro de una estructura `readonly` modifique el estado de la misma.

> **NOTE**
> En una estructura `readonly`, un miembro de datos de un tipo de referencia mutable puede seguir mutando su propio estado. Por ejemplo, no puede reemplazar una instancia de List<T>, pero puede agregarle nuevos elementos.

Miembros de instancia de `readonly`

A partir de C# 8.0, también puede usar el modificador `readonly` para declarar que un miembro de instancia no modifica el estado de una estructura. Si no puede declarar el tipo de estructura completa como `readonly`, use el modificador `readonly` para marcar los miembros de instancia que no modifican el estado de la estructura. En una estructura `readonly`, cada miembro de instancia es `readonly` de forma implícita.

Dentro de un miembro de instancia `readonly`, no se puede realizar la asignación a campos de instancia de la estructura. Pero un miembro `readonly` puede llamar a un miembro que no sea `readonly`. En ese caso, el compilador crea una copia de la instancia de la estructura y llama al miembro que no es `readonly` en esa copia. Como resultado, la instancia de la estructura original no se modifica.

Normalmente, se aplica el modificador `readonly` a los siguientes tipos de miembros de instancia:

- Métodos:

  ```
  public readonly double Sum()
  {
      return X + Y;
  }
  ```

  También puede aplicar el modificador `readonly` a los métodos que invalidan los

métodos declarados en System.Object:

```
public readonly override string ToString() => $"({X}, {Y})";
```

- Propiedades e indizadores:

```
private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}
```

Si tiene que aplicar el modificador `readonly` a los dos descriptores de acceso de una propiedad o un indizador, aplíquelo en la declaración de la propiedad o el indizador.

> **NOTE**
> El compilador declara un descriptor de acceso `get` de una propiedad implementada automáticamente como `readonly`, con independencia de la presencia del modificador `readonly` en la declaración de una propiedad.

No se puede aplicar el modificador `readonly` a los miembros estáticos de un tipo de estructura.

Es posible que el compilador use el modificador `readonly` para optimizaciones de rendimiento. Para más información, consulte Escritura de código C# seguro y eficaz.

Limitaciones del diseño de un tipo de estructura

Al diseñar un tipo de estructura, tiene las mismas funciones que con un tipo de clase, con las siguientes excepciones:

- No se puede declarar un constructor sin parámetros. Cada tipo de estructura ya proporciona un constructor sin parámetros implícito que genera el valor predeterminado del tipo.
- No se puede inicializar una propiedad o un campo de instancia en su declaración. Pero se puede inicializar un campo static o const, o una propiedad estática en su declaración.
- Un constructor de un tipo de estructura debe inicializar todos los campos de instancia del tipo.
- Un tipo de estructura no puede heredar de otro tipo de estructura o clase ni puede ser la base de una clase. Pero un tipo de estructura puede implementar interfaces.
- No se puede declarar un finalizador dentro de un tipo de estructura.

Creación de instancias de un tipo de estructura

En C#, debe inicializar una variable declarada antes de poder usarla. Como una variable de tipo de estructura no puede ser `null` (a menos que sea una variable de un tipo de valor que admite valores NULL), tendrá que crear instancias de una instancia del tipo correspondiente. Existen varias formas de hacerlo.

Normalmente, para crear una instancia de un tipo de estructura, se llama a un constructor

adecuado con el operador `new`. Todos los tipos de estructura tienen al menos un constructor. Se trata de un constructor sin parámetros implícito, que genera el valor predeterminado del tipo. También puede usar una expresión de valor predeterminado para generar el valor predeterminado de un tipo.

Si se puede acceder a todos los campos de instancia de un tipo de estructura, también puede crear una instancia de él sin el operador `new`. En ese caso, debe inicializar todos los campos de instancia antes del primer uso de la instancia. En el siguiente ejemplo se muestra cómo hacerlo:

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"({p.x}, {p.y})");  // output: (3, 4)
    }
}
```

En el caso de los tipos de valor integrados, use los literales correspondientes para especificar un valor del tipo.

Pasar variables de tipo de estructura por referencia

Cuando se pasa una variable de tipo de estructura a un método como argumento o se devuelve un valor de tipo de estructura a partir de un método, se copia toda la instancia de un tipo de estructura. Esto puede afectar al rendimiento del código en escenarios de alto rendimiento que impliquen tipos de estructura grandes. Puede evitar la copia de valores si pasa una variable de tipo de estructura por referencia. Use los modificadores de parámetro de método `ref`, `out` o `in` para indicar que un argumento se debe pasar por referencia. Use valores devueltos de tipo ref para devolver el resultado de un método por referencia. Para más información, consulte Escritura de código C# seguro y eficaz.

Estructura `ref`

A partir de C# 7.2, puede usar el modificador `ref` en la declaración de un tipo de estructura. Las instancias de un tipo de estructura `ref` se asignan en la pila y no pueden escapar al montón administrado. Para asegurarse de eso, el compilador limita el uso de tipos de estructura `ref` de la manera siguiente:

- Una estructura `ref` no puede ser el tipo de elemento de una matriz.
- Una estructura `ref` no puede ser un tipo declarado de un campo de una clase o una estructura que no sea `ref`.
- Una estructura `ref` no puede implementar interfaces.
- En una estructura `ref` no se puede aplicar una conversión boxing a System.ValueType ni System.Object.
- Una estructura `ref` no puede ser un argumento de tipo.

- Una `ref` variable de estructura no se puede capturar mediante una expresión lambda o una función local.
- Una variable de estructura `ref` no se puede usar en un método `async`. Aunque se pueden usar variables de estructura `ref` en métodos sincrónicos, como, por ejemplo, en los que devuelven Task o Task<TResult>.
- Una variable de estructura `ref` no se puede usar en iteradores.

Normalmente, se define un tipo de estructura `ref` cuando se necesita un tipo que también incluye miembros de datos de tipos de estructura `ref`:

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

Para declarar una estructura `ref` como `readonly`, combine los modificadores `readonly` y `ref` en la declaración de tipos (el modificador `readonly` debe ir delante del modificador `ref`):

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

En .NET, System.Span<T> y System.ReadOnlySpan<T> son ejemplos de una estructura `ref`.

Conversiones

Para cualquier tipo de estructura (excepto los tipos de estructura `ref`), existen conversiones boxing y unboxing a y desde los tipos System.ValueType y System.Object. También existen conversiones boxing y unboxing entre un tipo de estructura y cualquier interfaz que implemente.

Especificación del lenguaje C#

Para más información, vea la sección Estructuras de la especificación del lenguaje C#.

Para obtener más información sobre de las características presentadas en C# 7.2 y versiones posteriores, vea las siguientes notas de propuesta de características:

- Estructuras readonly
- Miembros de instancia readonly
- Seguridad de tiempo de compilación para tipos similares a ref

Vea también

- Referencia de C#

- Instrucciones de diseño: elección entre clase y estructura
- Instrucciones de diseño: diseño de estructuras
- Clases y estructuras

Tipos de valor que admiten valores NULL (referencia de C#)

04/05/2020 • 14 minutes to read • Edit Online

Un *tipo de valor que admite un valor NULL* `T?` representa todos los valores de su tipo de valor subyacente `T` y un valor NULL adicional. Por ejemplo, puede asignar cualquiera de los tres valores siguientes a una variable `bool?`: `true`, `false` o `null`. Un tipo de valor subyacente `T` no puede ser un tipo de valor que acepte valores NULL por sí mismo.

> **NOTE**
>
> C# 8.0 presenta la característica de tipos de referencia que admiten un valor NULL. Para más información, consulte Tipos de referencia que admiten un valor NULL. Los tipos de valor que admiten valores NULL están disponibles a partir de C# 2.

Todos los tipos que admiten valores NULL son instancias de la estructura System.Nullable<T> genérica. Puede hacer referencia a un tipo de valor que admite valores NULL con un tipo subyacente `T` en cualquiera de las formas intercambiables siguientes: `Nullable<T>` o `T?`.

Normalmente, los tipos de valor que admiten valores NULL se usan cuando es necesario representar el valor indefinido de un tipo de valor subyacente. Por ejemplo, una variable booleana, o `bool`, solo puede ser `true` o `false`. Sin embargo, en algunas aplicaciones, un valor de variable puede estar sin definir o faltar. Por ejemplo, un campo de base de datos puede contener `true` o `false`, o puede no contener ningún valor, es decir, `NULL`. Puede usar el tipo `bool?` en ese escenario.

Declaración y asignación

Como un tipo de valor se puede convertir de forma implícita al tipo de valor que admite valores NULL correspondiente, un valor se puede asignar a un tipo que admite valores NULL como se haría para su tipo de valor subyacente. También se puede asignar el valor `null`. Por ejemplo:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

El valor predeterminado de un tipo de valor que admite valores NULL se representa como `null`, es decir, es una instancia cuya propiedad Nullable<T>.HasValue devuelve `false`.

Examen de una instancia de tipo de valor que admite valores NULL

A partir C# 7.0, puede usar el operador `is` con un patrón de tipo para examinar una instancia de un tipo de valor que admite valores NULL para `null` y recuperar un valor de un tipo subyacente:

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

Siempre puede usar las siguientes propiedades de solo lectura para examinar y obtener un valor de una variable de tipo de valor que admite valores NULL:

- Nullable<T>.HasValue indica si una instancia de un tipo que admite valores NULL tiene un valor de su tipo subyacente.
- Nullable<T>.Value obtiene el valor de un tipo subyacente si HasValue es `true`. Si HasValue es `false`, la propiedad Value inicia una excepción InvalidOperationException.

En el ejemplo siguiente se usa la propiedad `HasValue` para comprobar si la variable contiene un valor antes de mostrarlo:

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

También se puede comparar una variable de un tipo de valor que admite valores NULL con `null` en lugar de usar la propiedad `HasValue`, como se muestra en el ejemplo siguiente:

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

Conversión de un tipo de valor que admite un valor NULL a un tipo subyacente

Si desea asignar un valor de un tipo que admite valores NULL a una variable de tipo de valor que no admite valores NULL, puede que tenga que especificar el valor que se va a asignar en lugar de `null`. Use el operador de fusión de NULL `??` para ello (también puede usar el método Nullable<T>.GetValueOrDefault(T) con el mismo fin):

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // output: d is -1
```

Si desea utilizar el valor predeterminado del tipo de valor subyacente en lugar de `null`, use el método Nullable<T>.GetValueOrDefault().

También puede convertir de forma explícita un tipo de valor que admite valores NULL en uno que no los admite, como se indica en el ejemplo siguiente:

```
int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

En tiempo de ejecución, si el valor de un tipo de valor que admite un valor NULL es `null`, la conversión explícita inicia una InvalidOperationException.

Un tipo de valor que no admite valores NULL `T` se convierte implícitamente al tipo que admite valores NULL `T?` correspondiente.

Operadores de elevación

Los operadores unarios y binarios predefinidos o los operadores sobrecargados que admite un tipo de valor `T` también se admiten en el tipo de valor que admite un valor NULL `T?` correspondiente. Estos operadores, también conocidos como *operadores de elevación*, generan un valor `null` si uno o los dos operandos son `null`; de lo contrario, el operador usa los valores contenidos de sus operandos para calcular el resultado. Por ejemplo:

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c;  // a is 110
a = a + b;  // a is null
```

> **NOTE**
>
> Para el tipo `bool?`, los operadores predefinidos `&` y `|` no siguen las reglas descritas en esta sección: el resultado de una evaluación de operador puede ser distinto de NULL incluso si uno de los operandos es `null`. Para más información, consulte la sección Operadores lógicos booleanos que aceptan valores NULL del artículo Operadores lógicos booleanos.

En el caso de los operadores de comparación `<`, `>`, `<=` y `>=`, si uno o ambos operandos son `null`, el resultado es `false`; de lo contrario, se comparan los valores contenidos de los operandos. No asuma que, como una comparación determinada (por ejemplo, `<=`) devuelve `false`, la comparación contraria (`>`) devolverá `true`. En el ejemplo siguiente se muestra que 10 no es

- mayor ni igual que `null`,

- ni es menor que `null`.

```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

En el caso de los operadores de igualdad `==`, si ambos operandos son `null`, el resultado es `true`; si solo uno de los operandos es `null`, el resultado es `false`; de lo contrario, se comparan los valores contenidos de los operandos.

En el caso de los operadores de desigualdad `!=`, si ambos operandos son `null`, el resultado es `false`; si solo uno de los operandos es `null`, el resultado es `true`; de lo contrario, se comparan los valores contenidos de los operandos.

Si existe una conversión definida por el usuario entre dos tipos de valor, esa misma conversión también se puede usar entre los correspondientes tipos que aceptan valores NULL.

Conversión boxing y conversión unboxing

La conversión boxing de una instancia de un tipo de valor que acepta valores NULL `T?` se realiza de la siguiente manera:

- Si HasValue devuelve `false`, se genera la referencia nula.
- Si HasValue devuelve `true`, se aplica la conversión boxing al correspondiente valor del tipo de valor subyacente `T`, no a la instancia de Nullable<T>.

Se puede aplicar conversión unboxing a un tipo de valor `T` con conversión boxing al tipo `T?` que acepta valores NULL correspondiente, como se muestra en el ejemplo siguiente:

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41
```

Identificación de tipos que admiten valores NULL

El ejemplo siguiente muestra cómo determinar si una instancia System.Type representa un tipo de valor

construido que admite valores NULL, es decir, el tipo System.Nullable<T> con un parámetro de tipo especificado `T`:

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type
```

Como se muestra en el ejemplo, se usa el operador typeof para crear una instancia System.Type.

Si quiere determinar si una instancia es de un tipo de valor que admite un valor NULL, no use el método Object.GetType para obtener una instancia de Type para probarla con el código anterior. Cuando se llama al método Object.GetType en una instancia de un tipo de valor que admite un valor NULL, se aplica la conversión boxing a la instancia para convertirla en Object. Como la conversión boxing de una instancia que no es NULL de un tipo de valor que admite valores NULL equivale a la conversión boxing de un valor del tipo subyacente, GetType devuelve una instancia Type que representa el tipo de valor subyacente que admite valores NULL:

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

No use el operador is para determinar si una instancia es de un tipo de valor que admite valores NULL. Como se muestra en el ejemplo siguiente, no se pueden distinguir los tipos de instancias de un tipo de valor que admite valores NULL y su tipo subyacente mediante el operador `is`:

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

Se puede usar el código que se presenta en el ejemplo siguiente para determinar si una instancia es de un tipo de valor que admite un valor NULL:

```
int? a = 14;
Console.WriteLine(IsOfNullableType(a));  // output: True

int b = 17;
Console.WriteLine(IsOfNullableType(b));  // output: False

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}
```

> **NOTE**
>
> Los métodos que se describen en esta sección no son aplicables en el caso de los tipos de referencia nula.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Nullable types (Tipos que aceptan valores NULL [Guía de programación de C#])
- Operadores de elevación
- Conversiones implícitas que aceptan valores NULL
- Conversiones explícitas que aceptan valores NULL
- Operadores de conversión de elevación

Vea también

- Referencia de C#
- ¿Qué significa exactamente "elevado"?
- System.Nullable<T>
- System.Nullable
- Nullable.GetUnderlyingType
- Tipos de referencia que aceptan valores null

Tipos de referencia (referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Hay dos clases de tipos en C#: tipos de referencia y tipos de valor. Las variables de tipos de referencia almacenan referencias en sus datos (objetos), mientras que las variables de tipos de valor contienen directamente los datos. Con los tipos de referencia, dos variables pueden hacer referencia al mismo objeto y, por lo tanto, las operaciones en una variable pueden afectar al objeto al que hace referencia la otra variable. Con los tipos de valor, cada variable tiene su propia copia de los datos, y no es posible que las operaciones en una variable afecten a la otra (excepto en el caso de las variables de parámetro in, ref y out, consulte el modificador de parámetro in, ref y out).

Las palabras clave siguientes se usan para declarar tipos de referencia:

- class
- interface
- delegate

C# también proporciona los siguientes tipos de referencia integrados:

- dynamic
- object
- string

Vea también

- Referencia de C#
- Palabras clave de C#
- Tipos de puntero
- Tipos de valor

Tipos de referencia integrados (referencia de C#)

18/03/2020 • 12 minutes to read • Edit Online

C# tiene un número de tipos de referencia integrados. Tienen palabras clave u operadores que son sinónimos para un tipo en la biblioteca de .NET.

Tipo object

El tipo `object` es un alias de System.Object en .NET. En el sistema de tipos unificado de C#, todos los tipos, los predefinidos y los definidos por el usuario, los tipos de referencia y los tipos de valores, heredan directa o indirectamente de System.Object. Puede asignar valores de cualquier tipo a las variables de tipo `object`. Cualquier variable `object` puede asignarse a su valor predeterminado con el literal `null`. Cuando una variable de un tipo de valor se convierte en objeto, se dice que se aplica la *conversión boxing*. Cuando una variable de tipo `object` se convierte en un tipo de valor, se dice que se *aplica la conversión unboxing*. Para obtener más información, vea Conversión boxing y unboxing.

Tipo string

El tipo `string` representa una secuencia de cero o más caracteres Unicode. `string` es un alias de System.String en .NET.

Aunque `string` es un tipo de referencia, se definen los operadores de igualdad `==` y `!=` para comparar los valores de los objetos `string`, no las referencias. Esto hace que las pruebas de igualdad entre cadenas sean más intuitivas. Por ejemplo:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

Se muestra "True" y luego "False" porque el contenido de las cadenas es equivalente, pero `a` y `b` no hacen referencia a la misma instancia de cadena.

El operador + concatena cadenas:

```
string a = "good " + "morning";
```

Crea un objeto de cadena que contiene "good morning".

Las cadenas son *inmutables*: el contenido de un objeto de cadena no se puede modificar una vez creado el objeto, aunque la sintaxis parezca indicar que se puede hacer. Por ejemplo, al escribir este código, el compilador crea en realidad otro objeto de cadena para almacenar la nueva secuencia de caracteres, y este nuevo objeto se asigna a `b`. La memoria que se había asignado para `b` (cuando contiene la cadena "h") es entonces apto para la recolección de elementos.

```
string b = "h";
b += "ello";
```

El operador `[]` puede usarse para el acceso de solo lectura a determinados caracteres de una cadena. Los valores válidos comienzan por `0` y deben ser menores que la longitud de la cadena:

```
string str = "test";
char x = str[2];  // x = 's';
```

De igual manera, el operador `[]` también puede usarse para recorrer en iteración cada carácter en una cadena:

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
  Console.Write(str[i] + " ");
}
// Output: t e s t
```

Los literales de cadena son del tipo `string` y se pueden escribir de dos formas, entre comillas y entre `@`. Los literales de cadena se incluyen entre comillas dobles ("):

```
"good morning"  // a string literal
```

Los literales de cadena pueden contener cualquier literal de carácter. Se incluyen secuencias de escape. En el ejemplo siguiente se usa una secuencia de escape `\\` para la barra diagonal inversa, `\u0066` para la letra f y `\n` para la nueva línea.

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
//  F
```

NOTE

El código de escape `\udddd` (donde `dddd` es un número de cuatro dígitos) representa el carácter Unicode U+`dddd`. También se reconocen los códigos de escape Unicode de 8 dígitos: `\Udddddddd`.

Los literales de cadena textual empiezan por `@` y también se incluyen entre comillas dobles. Por ejemplo:

```
@"good morning"  // a string literal
```

La ventaja de las cadenas textuales es que las secuencias de escape *no* se procesan, lo que facilita la escritura de, por ejemplo, un nombre de archivo Windows completo:

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

Para incluir un signo de comillas dobles en una cadena @-quoted, duplíquelo:

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

Tipo delegate

La declaración de un tipo delegado es similar a una firma de método. Tiene un valor devuelto y un número cualquiera de parámetros de cualquier tipo:

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

En. NET, los tipos `System.Action` y `System.Func` proporcionan definiciones genéricas para muchos delegados comunes. Es probable que no sea necesario definir nuevos tipos delegados personalizados. En su lugar, puede crear instancias de los tipos genéricos proporcionados.

Un `delegate` es un tipo de referencia que puede usarse para encapsular un método con nombre o anónimo. Los delegados son similares a los punteros de función en C++; pero son más seguros y proporcionan mayor seguridad de tipos. Para las aplicaciones de delegados, vea Delegados y Delegados genéricos. Los delegados son la base de los eventos. Se pueden crear instancias de un delegado asociándolo a un método con nombre o anónimo.

Para crear instancias del delegado debe usarse un método o una expresión lambda que tenga un tipo de valor devuelto y parámetros de entrada compatibles. Para obtener más información sobre el grado de variación permitida en la firma de método, vea Varianza en delegados. Para el uso con métodos anónimos, el delegado y el código que se van a asociar se declaran juntos.

Tipo dynamic

El tipo `dynamic` indica el uso de la variable y las referencias a su comprobación de tipos en el tiempo de compilación de omisión de miembros. En su lugar, se resuelven estas operaciones en tiempo de ejecución. El tipo `dynamic` simplifica el acceso a las API de COM como las API de automatización de Office, a API dinámicas como las bibliotecas de IronPython, y a Document Object Model (DOM) HTML.

El tipo `dynamic` se comporta como el tipo `object` en la mayoría de las circunstancias. En concreto, se puede convertir cualquier expresión no NULL para el tipo `dynamic`. El tipo `dynamic` se diferencia de `object` en que el compilador no resuelve o no comprueba el tipo de las operaciones que contienen expresiones de tipo `dynamic`. El compilador empaqueta información sobre la operación y esa información se usa después para evaluar la operación en tiempo de ejecución. Como parte del proceso, las variables de tipo `dynamic` están compiladas en las variables de tipo `object`. Por consiguiente, el tipo `dynamic` solo existe en tiempo de compilación, no en tiempo de ejecución.

En el siguiente ejemplo se contrasta una variable de tipo `dynamic` con una variable de tipo `object`. Para comprobar el tipo de cada variable en tiempo de compilación, coloque el puntero del mouse sobre `dyn` u `obj` en las instrucciones `WriteLine`. Copie el código siguiente en un editor donde IntelliSense esté disponible. IntelliSense muestra **dynamic** para `dyn` y **object** para `obj`.

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

Las instrucciones WriteLine muestran los tipos en tiempo de ejecución de `dyn` y `obj`. En ese punto, ambos tienen el mismo tipo, entero. Se produce el siguiente resultado:

```
System.Int32
System.Int32
```

Para ver la diferencia entre `dyn` y `obj` en tiempo de compilación, agregue las dos líneas siguientes entre las declaraciones y las instrucciones `WriteLine` en el ejemplo anterior.

```
dyn = dyn + 3;
obj = obj + 3;
```

Un error del compilador se notifica para el intento de suma de un entero y un objeto en la expresión `obj + 3`. En cambio, no se notifica ningún error para `dyn + 3`. En tiempo de compilación no se comprueba la expresión que contiene `dyn` porque el tipo de `dyn` es `dynamic`.

El ejemplo siguiente usa `dynamic` en varias declaraciones. El método `Main` también contrasta la comprobación de tipo en tiempo de compilación con la comprobación de tipo en tiempo de ejecución.

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because exampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to exampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

Vea también

- Referencia de C#
- Palabras clave de C#
- Eventos
- Uso de tipo dinámico
- Procedimientos recomendados para el uso de cadenas
- Operaciones básicas de cadenas
- Creación de cadenas nuevas
- Operadores de conversión y prueba de tipos
- Procedimiento para convertir de forma segura mediante la coincidencia de patrones y los

operadores is y as

- Tutorial: Crear y usar objetos dinámicos (C# y Visual Basic)
- System.Object
- System.String
- System.Dynamic.DynamicObject

class (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

Las clases se declaran mediante la palabra clave `class`, como se muestra en el siguiente ejemplo:

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

Comentarios

Solo la herencia simple se permite en C#. En otras palabras, una clase puede heredar la implementación solo de una clase base. En cambio, una clase puede implementar más de una interfaz. En la tabla siguiente se muestran ejemplos de herencia de clases e implementación de interfaces:

HERENCIA	EJEMPLO
None	`class ClassA { }`
Single	`class DerivedClass : BaseClass { }`
Ninguna, implementa dos interfaces	`class ImplClass : IFace1, IFace2 { }`
Única, implementa una interfaz	`class ImplDerivedClass : BaseClass, IFace1 { }`

Las clases que se declaran directamente dentro de un espacio de nombres, que no están anidadas dentro de otras clases, pueden ser de tipo public o internal. De forma predeterminada, las clases son `internal`.

Los miembros de clase, incluidas las clases anidadas, pueden ser public, protected internal, protected, internal, private o private protected. De forma predeterminada, los miembros son `private`.

Para obtener más información, consulte Modificadores de acceso.

Puede declarar clases genéricas que tengan parámetros de tipo. Para obtener más información, consulte Clases genéricas.

Una clase puede contener declaraciones de los miembros siguientes:

- Constructores
- Constantes
- Campos
- Finalizadores

- Métodos
- Propiedades
- Indizadores
- Operadores
- Eventos
- Delegados
- Clases
- Interfaces
- Tipos de estructura
- Tipos de enumeración

Ejemplo

En el ejemplo siguiente se muestra cómo declarar campos de clase, constructores y métodos. También se muestra la creación de instancias de objeto y la impresión de datos de instancias. En este ejemplo, se declaran dos clases. La primera clase, `Child`, contiene dos campos privados (`name` y `age`), dos constructores públicos y un método público. La segunda clase, `StringTest`, se usa para contener `Main`.

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/
```

Comentarios

Observe que en el ejemplo anterior solo se puede acceder a los campos privados (`name` y `age`) a través del método público de la clase `Child`. Por ejemplo, no puede imprimir el nombre de la clase child, desde el método `Main`, mediante una instrucción como esta:

```
Console.Write(child1.name);   // Error
```

El acceso a miembros privados de `Child` desde `Main` solo sería posible si `Main` fuera un miembro de la clase.

Los tipos declarados dentro de una clase sin un modificador de acceso adoptan el valor predeterminado de `private`, por lo que los miembros de datos de este ejemplo seguirían siendo `private` si se quitara la palabra clave.

Por último, tenga en cuenta que, para el objeto creado mediante el constructor sin parámetros (`child3`), el campo `age` se ha inicializado en cero de forma predeterminada.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Tipos de referencia

interface (Referencia de C#)

29/06/2020 • 5 minutes to read • Edit Online

Una interfaz define un contrato. Cualquier `class` o `struct` que implemente ese contrato debe proporcionar una implementación de los miembros definidos en la interfaz. A partir de C# 8.0, una interfaz puede definir una implementación predeterminada de miembros. También puede definir miembros `static` para proporcionar una única implementación de funcionalidad común.

En el ejemplo siguiente, la clase `ImplementationClass` debe implementar un método denominado `SampleMethod` que no tiene ningún parámetro y devuelve `void`.

Para obtener más información y ejemplos, vea Interfaces.

Ejemplo

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

Una interfaz puede ser un miembro de un espacio de nombres o una clase. Una declaración de interfaz puede contener declaraciones (firmas sin ninguna implementación) de los miembros siguientes:

- Métodos
- Propiedades
- Indizadores
- Eventos

Normalmente, estas declaraciones de miembros anteriores no contienen ningún cuerpo. A partir de C# 8.0, un miembro de interfaz puede declarar un cuerpo. Esto se conoce como *implementación predeterminada*. Los miembros con cuerpos permiten que la interfaz proporcione una implementación "predeterminada" de las clases y las estructuras que no proporcionan una implementación de invalidación. Además, a partir de C# 8.0, una interfaz puede incluir:

- Constantes
- Operadores

- Constructor estático.
- Tipos anidados
- Campos, métodos, propiedades, indizadores y eventos estáticos.
- Declaraciones de miembros con la sintaxis de implementación de interfaz explícita.
- Modificadores de acceso explícitos (el acceso predeterminado es `public`).

Es posible que las interfaces no contengan estado de instancia. Aunque los campos estáticos ahora están permitidos, los campos de instancia no se permiten en las interfaces. Las propiedades automáticas de instancia no se admiten en las interfaces, ya que declararían de forma implícita un campo oculto. Esta regla tiene un efecto sutil en las declaraciones de propiedad. En una declaración de interfaz, el código siguiente no declara una propiedad implementada automáticamente como hace en una `class` o un `struct`. En su lugar, declara una propiedad que no tiene una implementación predeterminada pero que se debe implementar en cualquier tipo que implemente la interfaz:

```
public interface INamed
{
  public string Name {get; set;}
}
```

Una interfaz puede heredar de una o varias interfaces base. Cuando una interfaz invalida un método implementado en una interfaz base, debe usar la sintaxis de implementación de interfaz explícita.

Cuando una lista de tipos base contiene una clase e interfaces base, la clase base debe aparecer primero en la lista.

Una clase que implementa una interfaz puede implementar explícitamente miembros de esa interfaz. A un miembro implementado explícitamente solo se puede tener acceso mediante una instancia de la interfaz, y no mediante una instancia de la clase. Además, solo se puede acceder a los miembros de interfaz predeterminados a través de una instancia de la interfaz.

Para obtener más información sobre la implementación de interfaz explícita, vea Implementación de interfaz explícita.

Ejemplo

En el ejemplo siguiente se muestra la implementación de una interfaz. En este ejemplo, la interfaz contiene la declaración de propiedad y la clase contiene la implementación. Cualquier instancia de una clase que implemente `IPoint` tiene las propiedades de entero `x` e `y`.

```
interface IPoint
{
   // Property signatures:
   int X
   {
      get;
      set;
   }

   int Y
   {
      get;
      set;
   }

   double Distance
   {
       get;
   }
}

class Point : IPoint
{
   // Constructor:
   public Point(int x, int y)
   {
      X = x;
      Y = y;
   }

   // Property implementation:
   public int X { get; set; }

   public int Y { get; set; }

   // Property implementation
   public double Distance =>
      Math.Sqrt(X * X + Y * Y);

}

class MainClass
{
   static void PrintPoint(IPoint p)
   {
      Console.WriteLine("x={0}, y={1}", p.X, p.Y);
   }

   static void Main()
   {
      IPoint p = new Point(2, 3);
      Console.Write("My Point: ");
      PrintPoint(p);
   }
}
// Output: My Point: x=2, y=3
```

especificación del lenguaje C#

Para obtener más información, vea la sección Interfaces de Especificación del lenguaje C# y la especificación de características de Miembros de interfaz predeterminados (C# 8.0)

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Tipos de referencia
- Interfaces
- Utilizar propiedades
- Utilizar indizadores

Tipos de referencia que aceptan valores NULL (referencia de C#)

27/04/2020 • 10 minutes to read • Edit Online

> **NOTE**
> En este artículo se tratan los tipos de referencia que aceptan valores NULL. También puede declarar tipos de valor que aceptan valores NULL.

Los tipos de referencia que aceptan valores NULL están disponibles a partir de C# 8.0, en un código que ha participado en un *contexto compatible con valores NULL*. Los tipos de referencia que aceptan valores NULL, las advertencias de análisis estático NULL y el operador null-forgiving son características de lenguaje opcionales. Todas están desactivadas de forma predeterminada. Un *contexto que admite valores NULL* se controla en el nivel de proyecto mediante la configuración de compilación o en el código que usa pragmas.

En un contexto compatible con valores NULL:

- Una variable de un tipo de referencia `T` se debe inicializar con un valor distinto de NULL y nunca puede tener asignado un valor que sea `null`.
- Una variable de un tipo de referencia `T?` se puede inicializar con `null` o asignarle `null`, pero debe comprobarse con `null` antes de desreferenciarla.
- Una variable `m` de tipo `T?` se considera que no es NULL cuando se aplica el operador null-forgiving, como en `m!`.

Las distinciones entre un tipo de referencia que no acepta valores NULL `T` y un tipo de referencia que acepta valores NULL `T?` se aplican mediante la interpretación del compilador de las reglas anteriores. Una variable de tipo `T` y una variable de tipo `T?` se representan mediante el mismo tipo .NET. En el ejemplo siguiente se declara una cadena que no acepta valores NULL y una cadena que acepta valores NULL y luego se usa el operador null-forgiving para asignar un valor a una cadena que no acepta valores NULL:

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

Las variables `notNull` y `nullable` se ambas representan con el tipo String. Dado que los tipos que no aceptan valores NULL y que aceptan valores NULL se almacenan como el mismo tipo, hay varias ubicaciones en las que no se permite el uso de un tipo de referencia que acepte valores NULL. En general, un tipo de referencia que acepta valores NULL no se puede usar como clase base o interfaz implementada. No se puede usar un tipo de referencia que acepte valores NULL en ninguna expresión de creación de objetos o de expresión de prueba de tipos. Un tipo de referencia que acepta valores NULL no puede ser el tipo de una expresión de acceso a miembros. En los siguientes ejemplos se muestran estas construcciones:

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

Referencias que aceptan valores NULL y análisis estático

Los ejemplos de la sección anterior muestran la naturaleza de los tipos de referencia que aceptan valores NULL. Los tipos de referencia que aceptan valores NULL no son nuevos tipos de clase, sino anotaciones en tipos de referencia existentes. El compilador utiliza esas anotaciones para ayudarle a encontrar posibles errores de referencia nula en el código. No hay ninguna diferencia en tiempo de ejecución entre un tipo de referencia que no acepta valores NULL y un tipo de referencia que acepta valores NULL. El compilador no agrega ninguna comprobación de tiempo de ejecución para los tipos de referencia que no aceptan valores NULL. Las ventajas radican en el análisis en tiempo de compilación. El compilador genera advertencias que le ayudan a encontrar y corregir posibles errores nulos en el código. Declare su intención y el compilador le advierte cuando el código infringe dicha intención.

En un contexto habilitado para valores NULL, el compilador realiza un análisis estático de las variables de cualquier tipo de referencia, tanto si admiten valores NULL como si no aceptan valores NULL. El compilador realiza un seguimiento del estado NULL de cada variable de referencia como *no NULL* o *quizás NULL*. El estado predeterminado de una referencia que no acepta valores NULL es *no NULL*. El estado predeterminado de una referencia que acepta valores NULL es *quizás NULL*.

Los tipos de referencia que no aceptan valores NULL siempre deben ser seguros para desreferenciar porque su estado NULL es *no NULL*. Para aplicar esa regla, el compilador emite advertencias si un tipo de referencia que no acepta valores NULL no se inicializa en un valor no NULL. Las variables locales deben asignarse allí donde se declaran. Cada constructor debe asignar todos los campos, ya sea en el cuerpo, en un constructor llamado o con un inicializador de campo. El compilador emite advertencias si una referencia que no acepta valores NULL se asigna a una referencia cuyo estado es *quizás NULL*. Sin embargo, dado que una referencia que no acepta valores NULL es *no NULL*, no se emite ninguna advertencia cuando se desreferencian esas variables.

Los tipos de referencia que aceptan valores NULL se pueden inicializar o asignar a `null`. Por lo tanto, el análisis estático debe determinar que una variable es *no NULL* antes de desreferenciarla. Si se determina que una referencia que acepta valores NULL es *quizás NULL*, no se puede asignar a una variable de referencia que no acepte valores NULL. La consulta siguiente muestra ejemplos de estas advertencias:

```
public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! short description not initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string? details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference possible null
        {
            return shortDescription;
        } else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        } else if (detailedDescription.Length > 0) // OK, optionalGreeting can't be null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}
```

En el fragmento de código siguiente se muestra dónde el compilador emite advertencias al utilizar esta clase:

```
string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static analysis knows greeting maybe
null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");
```

En los ejemplos anteriores se muestra el análisis estático del compilador para determinar el estado NULL de las variables de referencia. El compilador aplica las reglas de lenguaje a las asignaciones y comprobaciones de valores NULL para informar de su análisis. El compilador no puede hacer suposiciones sobre la semántica de métodos o propiedades. Si llama a métodos que realizan comprobaciones de valores NULL, el compilador no puede saber que estos métodos afectan al estado NULL de una variable. Hay una serie de atributos que puede agregar a las API para informar al compilador sobre la semántica de los argumentos y los valores devueltos. Estos atributos se han aplicado a muchas API comunes de las bibliotecas de .NET Core. Por ejemplo, IsNullOrEmpty se ha actualizado y el compilador interpreta correctamente ese método como una comprobación de valores NULL. Para obtener más información sobre los atributos que se aplican al análisis estático de estado

NULL, consulte el artículo sobre atributos que aceptan valores NULL.

Establecimiento del contexto que acepta valores NULL

Hay dos formas de controlar el contexto que acepta valores NULL. En el nivel de proyecto, puede agregar la configuración del proyecto `<Nullable>enable</Nullable>`. En un archivo C# de código fuente único, puede agregar el pragma `#nullable enable` para habilitar el contexto que acepta valores NULL. Consulte el artículo sobre cómo establecer una estrategia que acepte valores NULL.

Especificación del lenguaje C#

Para más información, consulte las propuestas siguientes de la especificación del lenguaje C#:

- Tipos de referencia que aceptan valores null
- Borrador de especificación de tipos de referencia que aceptan valores NULL

Vea también

- Referencia de C#
- Tipos de valores que aceptan valores NULL

void (referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Use `void` como el tipo de valor devuelto de un método (o una función local) para especificar que el método no devuelve un valor.

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

También puede usar `void` como un tipo de referente para declarar un puntero a un tipo desconocido. Para obtener más información, vea Tipos de puntero.

No se puede usar `void` como el tipo de una variable.

Vea también

- Referencia de C#
- System.Void

var (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

A partir de Visual C# 3.0, las variables que se declaran en el ámbito de método pueden tener un "tipo" `var` implícito. Una variable local con tipo implícito es fuertemente tipada exactamente igual que si hubiera declarado el tipo, solo que en este caso es el compilador el que lo determina. Las dos declaraciones siguientes de `i` tienen una función equivalente:

```
var i = 10; // Implicitly typed.
int i = 10; // Explicitly typed.
```

Para obtener más información, vea Variables locales con asignación implícita de tipos y Relaciones entre tipos en operaciones de consulta LINQ.

Ejemplo

En el siguiente ejemplo se muestran dos expresiones de consulta. En la primera expresión, el uso de `var` se permite pero no es necesario, ya que el tipo del resultado de la consulta se puede indicar explícitamente como `IEnumerable<string>`. En cambio, en la segunda expresión, `var` permite que el resultado sea una colección de tipos anónimos y solo el compilador puede tener acceso al nombre de ese tipo. El uso de `var` elimina la necesidad de crear una nueva clase para el resultado. Tenga en cuenta que, en el ejemplo 2, la variable de iteración `foreach``item` también debe tener tipo implícito.

```
// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
                where word[0] == 'g'
                select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
                where cust.City == "Phoenix"
                select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}
```

Vea también

- Referencia de C#

- Guía de programación de C#
- Variables locales con asignación implícita de tipos

Tipos integrados (referencia de C#)

29/06/2020 • 2 minutes to read • Edit Online

En la siguiente tabla se muestran los tipos de valor de C#:

PALABRA CLAVE DE TIPO DE C#	TIPO DE .NET
`bool`	System.Boolean
`byte`	System.Byte
`sbyte`	System.SByte
`char`	System.Char
`decimal`	System.Decimal
`double`	System.Double
`float`	System.Single
`int`	System.Int32
`uint`	System.UInt32
`long`	System.Int64
`ulong`	System.UInt64
`short`	System.Int16
`ushort`	System.UInt16

En la siguiente tabla se muestran los tipos de referencia integrados de C#:

PALABRA CLAVE DE TIPO DE C#	TIPO DE .NET
`object`	System.Object
`string`	System.String
`dynamic`	System.Object

En las tablas anteriores, cada palabra clave de tipo de C# de la columna ubicada a la izquierda es un alias del tipo de .NET correspondiente. Son intercambiables. Por ejemplo, en las declaraciones siguientes se declaran variables del mismo tipo:

```
int a = 123;
System.Int32 b = 123;
```

La palabra clave `void` representa la ausencia de un tipo. Se usa como el tipo de valor devuelto de un método que no devuelve un valor.

Vea también

- Referencia de C#
- Valores predeterminados de los tipos de C#

Tipos no administrados (referencia de C#)

30/03/2020 • 2 minutes to read • Edit Online

Un tipo es un **tipo no administrado** si es cualquiera de los siguientes tipos:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` o `bool`
- Cualquier tipo enum.
- Cualquier tipo pointer.
- Cualquier tipo de struct definido por el usuario que solo contenga campos de tipos no administrados, en C# 7.3 y versiones anteriores, no es un tipo construido (un tipo que incluya al menos un argumento de tipo)

A partir de C# 7.3, puede usar `unmanaged` constraint para especificar que un parámetro de tipo es un tipo no administrado que no acepta valores NULL y que no es de puntero.

A partir C# 8.0, un tipo de struct *construido* que solo contenga campos de tipos no administrados también es no administrado, como se muestra en el ejemplo siguiente:

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is {sizeof(T)} bytes");
    }
}
// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

Un struct genérico puede ser el origen de los tipos no administrados y de los tipos construidos no administrados. En el ejemplo anterior se define un struct `Coords<T>` genérico y se presentan los ejemplos de tipos construidos no administrados. El ejemplo de un tipo no administrado es `Coords<object>`. No está administrado porque tiene los campos del tipo `object`, que es no administrado. Si desea que *todos* los tipos construidos sean tipos no administrados, use la restricción `unmanaged` en la definición de un struct genérico:

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

Especificación del lenguaje C#

Para obtener más información, vea la sección Tipos de puntero de Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Tipos de puntero
- Tipos relacionados con el intervalo y la memoria
- sizeof (operador)
- stackalloc

Valores predeterminados de los tipos de C# (referencia de C#)

02/04/2020 • 2 minutes to read • Edit Online

En la tabla siguiente se muestran los valores predeterminados de los tipos de C#:

TIPO	VALOR PREDETERMINADO
Cualquier tipo de referencia	`null`
Cualquier tipo numérico entero integrado	0 (cero)
Cualquier tipo numérico de punto flotante integrado	0 (cero)
bool	`false`
char	`'\0'` (U+0000)
enum	Valor generado por la expresión `(E)0`, donde `E` es el identificador de enumeración.
struct	El valor generado al establecer todos los campos de tipo de valor en sus valores predeterminados y todos los campos de tipo de referencia en `null`.
Cualquier tipo de valor que acepta valores NULL	Instancia para la que la propiedad HasValue es `false` y la propiedad Value no está definida. Este valor predeterminado también se conoce con el valor *null* de un tipo de valor que acepta valores NULL.

Use el operador `default` para producir el valor predeterminado de un tipo, como se muestra en el ejemplo siguiente:

```
int a = default(int);
```

A partir de C# 7.1, se puede usar el literal `default` para inicializar una variable con el valor predeterminado de su tipo:

```
int a = default;
```

Para un tipo de valor, el constructor implícito sin parámetros también genera el valor predeterminado del tipo, como se muestra en el ejemplo siguiente:

```
var n = new System.Numerics.Complex();
Console.WriteLine(n);  // output: (0, 0)
```

En tiempo de ejecución, si la instancia de System.Type representa un tipo de valor, puede usar el método Activator.CreateInstance(Type) para invocar el constructor sin parámetros y obtener el valor predeterminado

del tipo.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Valores predeterminados
- Constructores predeterminados

Vea también

- Referencia de C#
- Constructores

Modificadores de acceso (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Los modificadores de acceso son palabras clave que se usan para especificar la accesibilidad declarada de un miembro o un tipo. En esta sección se presentan los cuatro modificadores de acceso:

- `public`
- `protected`
- `internal`
- `private`

Pueden especificarse los siguientes seis niveles de accesibilidad con los modificadores de acceso:

- `public`: el acceso no está restringido.
- `protected`: el acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora.
- `internal`: el acceso está limitado al ensamblado actual.
- `protected internal`: el acceso está limitado al ensamblado actual o a los tipos derivados de la clase contenedora.
- `private`: el acceso está limitado al tipo contenedor.
- `private protected`: el acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora que hay en el ensamblado actual.

En esta sección también se presenta lo siguiente:

- Niveles de accesibilidad: usar los cuatro modificadores de acceso para declarar seis niveles de accesibilidad.
- Dominio de accesibilidad: especifica en qué secciones del programa se puede hacer referencia a dicho miembro.
- Restricciones en el uso de niveles de accesibilidad: un resumen de las restricciones sobre usar niveles de accesibilidad declarados.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Palabras clave de acceso
- Modificadores

Niveles de accesibilidad (Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

Use los modificadores de acceso `public`, `protected`, `internal` o `private` para especificar uno de los siguientes niveles de accesibilidad declarada para miembros.

ACCESIBILIDAD DECLARADA	SIGNIFICADO
`public`	El acceso no está restringido.
`protected`	El acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora.
`internal`	El acceso está limitado al ensamblado actual.
`protected internal`	El acceso está limitado al ensamblado actual o a los tipos derivados de la clase contenedora.
`private`	El acceso está limitado al tipo contenedor.
`private protected`	El acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora que hay en el ensamblado actual. Disponible desde la versión C# 7.2.

Solo se permite un modificador de acceso para un miembro o tipo, excepto cuando se usan las combinaciones `protected internal` o `private protected`.

No se permiten modificadores de acceso en espacios de nombres. Los espacios de nombres no tienen restricciones de acceso.

En función del contexto en el que se produce una declaración de miembro, solo se permiten ciertas accesibilidades declaradas. Si no se especifica ningún modificador de acceso en una declaración de miembro, se usa una accesibilidad predeterminada.

Los tipos de nivel superior, que no están anidados en otros tipos, solo pueden tener una accesibilidad `internal` o `public`. La accesibilidad predeterminada para estos tipos es `internal`.

Los tipos anidados, que son miembros de otros tipos, pueden tener accesibilidades declaradas como se indica en la tabla siguiente.

MIEMBROS DE	ACCESIBILIDAD DE MIEMBRO PREDETERMINADA	ACCESIBILIDAD DECLARADA PERMITIDA DEL MIEMBRO
`enum`	`public`	Ninguna

MIEMBROS DE	ACCESIBILIDAD DE MIEMBRO PREDETERMINADA	ACCESIBILIDAD DECLARADA PERMITIDA DEL MIEMBRO

`class`	`private`	`public`
		`protected`
		`internal`
		`private`
		`protected internal`
		`private protected`

`interface`	`public`	Ninguna

`struct`	`private`	`public`
		`internal`
		`private`

La accesibilidad de un tipo anidado depende de su dominio de accesibilidad, que viene determinado por la accesibilidad declarada del miembro y el dominio de accesibilidad del tipo contenedor inmediato. Sin embargo, el dominio de accesibilidad de un tipo anidado no puede superar al del tipo contenedor.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Dominio de accesibilidad
- Restricciones en el uso de los niveles de accesibilidad
- Modificadores de acceso
- public
- private
- protected
- internal

Dominio de accesibilidad (Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

El dominio de accesibilidad de un miembro especifica en qué secciones del programa se puede hacer referencia a dicho miembro. Si el miembro está anidado dentro de otro tipo, su dominio de accesibilidad viene determinado por el nivel de accesibilidad del miembro y por el dominio de accesibilidad del tipo contenedor inmediato.

El dominio de accesibilidad de un tipo de nivel superior es por lo menos el texto del programa del proyecto en el que se declara. Es decir, el dominio incluye todos los archivos de origen de este proyecto. El dominio de accesibilidad de un tipo anidado es, al menos, el texto del programa del tipo en el que se declara. Es decir, el dominio es el cuerpo del tipo, que incluye todos los tipos anidados. El dominio de accesibilidad de un tipo anidado no puede superar nunca al del tipo contenedor. Estos conceptos se muestran en el siguiente ejemplo.

Ejemplo

Este ejemplo contiene un tipo de nivel superior, `T1`, y dos clases anidadas, `M1` y `M2`. Las clases contienen campos que tienen diferentes accesibilidades declaradas. En el método `Main`, a cada instrucción le sigue un comentario que indica el dominio de accesibilidad de cada miembro. Observe que las instrucciones que intentan hacer referencia a los miembros inaccesibles están marcadas con comentarios. Si quiere ver los errores generados por el compilador cuando se intenta hacer referencia a un miembro inaccesible, quite los comentarios de uno en uno.

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
```

```
class MainClass
{
    static void Main()
    {
        // Access is unlimited.
        T1.publicInt = 1;

        // Accessible only in current assembly.
        T1.internalInt = 2;

        // Error CS0122: inaccessible outside T1.
        // T1.privateInt = 3;

        // Access is unlimited.
        T1.M1.publicInt = 1;

        // Accessible only in current assembly.
        T1.M1.internalInt = 2;

        // Error CS0122: inaccessible outside M1.
        //    T1.M1.privateInt = 3;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.publicInt = 1;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.internalInt = 2;

        // Error CS0122: inaccessible outside M2.
        //    T1.M2.privateInt = 3;

        // Keep the console open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Niveles de accesibilidad
- Restricciones en el uso de los niveles de accesibilidad
- Modificadores de acceso
- public
- private
- protected
- internal

Restricciones en el uso de los niveles de accesibilidad (Referencia de C#)

11/05/2020 • 4 minutes to read • Edit Online

Cuando especifique un tipo en una declaración, compruebe si el nivel de accesibilidad de este depende del nivel de accesibilidad de un miembro o de otro tipo. Por ejemplo, la clase base directa debe ser al menos igual de accesible que la clase derivada. Las siguientes declaraciones producen un error del compilador porque la clase base `BaseClass` es menos accesible que `MyClass`:

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

En la tabla siguiente se resumen las restricciones en los niveles de accesibilidad declarados.

CONTEXTO	COMENTARIOS
Clases	La clase base directa de un tipo de clase debe ser al menos igual de accesible que el propio tipo de clase.
Interfaces	Las interfaces base explícitas de un tipo de interfaz deben ser al menos igual de accesibles que el propio tipo de interfaz.
Delegados	El tipo de valor devuelto y los tipos de parámetros de un tipo de delegado deben ser al menos igual de accesibles que el propio tipo de delegado.
Constantes	El tipo de una constante debe ser al menos igual de accesible que la propia constante.
Campos	El tipo de un campo debe ser al menos igual de accesible que el propio campo.
Métodos	El tipo de valor devuelto y los tipos de parámetros de un método deben ser al menos igual de accesibles que el propio método.
Propiedades	El tipo de una propiedad debe ser al menos igual de accesible que la misma propiedad.
Eventos	El tipo de un evento debe ser al menos igual de accesible que el propio evento.
Indizadores	Los tipos de parámetro y el tipo de un indexador deben ser al menos igual de accesibles que el propio indexador.
Operadores	El tipo de valor devuelto y los tipos de parámetro de un operador deben ser al menos igual de accesibles que el propio operador.

CONTEXTO	COMENTARIOS
Constructores	Los tipos de parámetro de un constructor deben ser al menos igual de accesibles que el propio constructor.

Ejemplo

El siguiente ejemplo contiene declaraciones erróneas de diferentes tipos. El comentario que sigue a cada declaración indica el error del compilador previsto.

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.Write("Compiled successfully");
    }
}
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Dominio de accesibilidad
- Niveles de accesibilidad
- Modificadores de acceso
- public
- private
- protected
- internal

internal (Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

La palabra clave `internal` es un modificador de acceso para tipos y miembros de tipo.

> Esta página trata sobre el modificador de acceso `internal`. La palabra clave `internal` también forma parte del modificador de acceso `protected internal`.

Solo se puede tener acceso a los tipos internos o los miembros desde los archivos del mismo ensamblado, como en este ejemplo:

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

Para obtener una comparación de `internal` con los demás modificadores de acceso, vea Niveles de accesibilidad y Modificadores de acceso.

Para más información sobre los ensamblados, consulte Ensamblados en .NET.

Un uso común del acceso interno se da en el desarrollo basado en componentes porque permite que un grupo de componentes cooperen de manera privada sin estar expuesto al resto del código de la aplicación. Por ejemplo, un marco para crear interfaces gráficas de usuario podría proporcionar las clases `Control` y `Form` que cooperan mediante miembros con acceso interno. Como estos miembros son internos, no se exponen al código que usa el marco de trabajo.

Es un error hacer referencia a un tipo o miembro con acceso interno fuera del ensamblado en el que se definió.

Ejemplo

Este ejemplo contiene dos archivos, `Assembly1.cs` y `Assembly1_a.cs`. El primer archivo contiene una clase base interna, `BaseClass`. En el segundo archivo, un intento de crear una instancia de `BaseClass` producirá un error.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // CS0122
   }
}
```

Ejemplo

En este ejemplo, use los mismos archivos usados en el ejemplo 1 y cambie el nivel de accesibilidad de `BaseClass` a `public`. Cambie también el nivel de accesibilidad del miembro `intM` a `internal`. En este caso, se puede crear una instancia de la clase, pero no se puede tener acceso al miembro interno.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

Especificación del lenguaje C#

Para obtener más información, vea la sección Accesibilidad declarada de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Niveles de accesibilidad
- Modificadores
- public
- private
- protected

private (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `private` es un modificador de acceso de miembro.

> Esta página trata sobre el modificador de acceso `private`. La palabra clave `private` también forma parte del modificador de acceso `private protected`.

El acceso privado es el nivel de acceso menos permisivo. Los miembros privados solo son accesibles dentro del cuerpo de la clase o el struct en el que se declaran, como en este ejemplo:

```
class Employee
{
    private int i;
    double d;   // private access by default
}
```

Los tipos anidados en el mismo cuerpo también pueden tener acceso a los miembros privados.

Hacer referencia a un miembro privado fuera de la clase o el struct en el que se declara es un error en tiempo de compilación.

Para obtener una comparación de `private` con los demás modificadores de acceso, vea Niveles de accesibilidad y Modificadores de acceso.

Ejemplo

En este ejemplo, la clase `Employee` contiene dos miembros de datos privados, `name` y `salary`. Como miembros privados, solo pueden tener acceso a ellos los métodos de miembro. Los métodos públicos denominados `GetName` y `Salary` se agregan para permitir un acceso controlado a los miembros privados. Se tiene acceso al miembro `name` a través de un método público, mientras que se tiene acceso al miembro `salary` a través de una propiedad pública de solo lectura. (Para obtener más información, vea Propiedades).

```
class Employee2
{
    private string name = "FirstName, LastName";
    private double salary = 100.0;

    public string GetName()
    {
        return name;
    }

    public double Salary
    {
        get { return salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //    string n = e.name;
        //    double s = e.salary;

        // 'name' is indirectly accessed via method:
        string n = e.GetName();

        // 'salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

especificación del lenguaje C#

Para obtener más información, vea la sección Accesibilidad declarada de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Niveles de accesibilidad
- Modificadores
- public
- protected
- internal

protected (Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

La palabra clave `protected` es un modificador de acceso de miembro.

> Esta página trata sobre el modificador de acceso `protected`. La palabra clave `protected` también forma parte de los modificadores de acceso `protected internal` y `private protected`.

Un miembro protegido es accesible dentro de su clase y por parte de instancias de clase derivadas.

Para obtener una comparación de `protected` con los demás modificadores de acceso, vea Niveles de accesibilidad.

Ejemplo

Un miembro protegido de una clase base es accesible en una clase derivada únicamente si el acceso se produce a través del tipo de clase derivada. Por ejemplo, vea el siguiente segmento de código:

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

La instrucción `a.x = 10` genera un error porque se ha creado en el método estático Main y no en una instancia de clase B.

Los miembros de estructura no se pueden proteger porque la estructura no puede heredarse.

Ejemplo

En este ejemplo, la clase `DerivedPoint` se deriva de `Point`. Por lo tanto, puede acceder a los miembros protegidos de la clase base directamente desde la clase derivada.

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

Si cambia los niveles de acceso de `x` y `y` a private, el compilador genera los mensajes de error:

```
'Point.y' is inaccessible due to its protection level.
```

```
'Point.x' is inaccessible due to its protection level.
```

especificación del lenguaje C#

Para obtener más información, vea la sección Accesibilidad declarada de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Niveles de accesibilidad
- Modificadores
- public
- private
- internal
- Security concerns for internal virtual keywords (Problemas de seguridad de palabras clave virtuales internas)

public (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `public` es un modificador de acceso para tipos y miembros de tipo. El acceso público es el nivel de acceso más permisivo. No hay ninguna restricción para el acceso a miembros públicos, como en este ejemplo:

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

Vea Modificadores de acceso y Niveles de accesibilidad para obtener más información.

Ejemplo

En el ejemplo siguiente, se declaran dos clases, `PointTest` y `MainClass`. Se obtiene acceso a los miembros públicos `x` e `y` de `PointTest` directamente desde `MainClass`.

```
class PointTest
{
    public int x;
    public int y;
}

class MainClass4
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

Si cambia el nivel de acceso `public` a private o protected, obtendrá el siguiente mensaje de error:

'PointTest.y' no es accesible debido a su nivel de protección.

especificación del lenguaje C#

Para obtener más información, vea la sección Accesibilidad declarada de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Modificadores de acceso
- Palabras clave de C#

- Modificadores de acceso
- Niveles de accesibilidad
- Modificadores
- private
- protected
- internal

protected internal (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La combinación de palabras claves `protected internal` es un modificador de acceso de miembro. Se puede obtener acceso a un miembro protected internal desde el ensamblado actual o desde tipos que se deriven de la clase contenedora. Para obtener una comparación de `protected internal` con los demás modificadores de acceso, vea Niveles de accesibilidad.

Ejemplo

Se puede obtener acceso a un miembro protected internal de una clase base desde cualquier tipo de ensamblado que lo contenga. También estará accesible en una clase derivada ubicada en otro ensamblado, pero solo si el acceso se produce a través de una variable del tipo de clase derivada. Por ejemplo, vea el siguiente segmento de código:

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
   protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

Este ejemplo contiene dos archivos, `Assembly1.cs` y `Assembly2.cs`. El primer archivo contiene una clase base interna, `BaseClass`, y otra clase, `TestAccess`. `BaseClass` posee un miembro protected internal (`myValue`), al que se obtiene acceso por medio del tipo `TestAccess`. En el segundo archivo, un intento de tener acceso a `myValue` a través de una instancia de `BaseClass` generará un error, mientras que un acceso a este miembro a través de una instancia de una clase derivada (`DerivedClass`) se realizará correctamente.

Los miembros de struct no pueden ser `protected internal`, porque los structs no se heredan.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Niveles de accesibilidad
- Modificadores
- public
- private
- internal
- Security concerns for internal virtual keywords (Problemas de seguridad de palabras clave virtuales internas)

private protected (Referencia de C#)

22/04/2020 • 3 minutes to read • Edit Online

La combinación de palabras claves `private protected` es un modificador de acceso de miembro. Los miembros private protected están accesibles para los tipos que se deriven de la clase contenedora, pero solo desde dentro del ensamblado correspondiente que lo contenga. Para obtener una comparación de `private protected` con los demás modificadores de acceso, vea Niveles de accesibilidad.

> **NOTE**
> El modificador de acceso `private protected` es válido en C# versión 7.2 y versiones posteriores.

Ejemplo

Se puede tener acceso a un miembro private protected de una clase base desde tipos derivados en el ensamblado que lo contiene solo si el tipo estático de la variable es el tipo de clase derivada. Por ejemplo, vea el siguiente segmento de código:

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

Este ejemplo contiene dos archivos, `Assembly1.cs` y `Assembly2.cs`. El primer archivo contiene una clase base pública, `BaseClass`, y un tipo derivado de ella, `DerivedClass1`. `BaseClass` posee un miembro private protected, `myValue`, al que `DerivedClass1` intenta tener acceso de dos maneras. El primer intento de acceso a

`myValue` a través de una instancia de `BaseClass` generará un error. En cambio, el intento de usarlo como un miembro heredado en `DerivedClass1` se realizará correctamente.

En el segundo archivo, un intento de tener acceso a `myValue` como un miembro heredado de `DerivedClass2` generará un error, ya que solo está accesible para los tipos derivados en Assembly1.

Si `Assembly1.cs` contiene un elemento InternalsVisibleToAttribute que asigna un nombre a `Assembly2`, la clase derivada `DerivedClass1` tendrá acceso a los miembros `private protected` declarados en `BaseClass`. `InternalsVisibleTo` hace que los miembros `private protected` sean visibles para las clases derivadas en otros ensamblados.

Los miembros de struct no pueden ser `private protected`, porque los structs no se heredan.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores de acceso
- Niveles de accesibilidad
- Modificadores
- public
- private
- internal
- Security concerns for internal virtual keywords (Problemas de seguridad de palabras clave virtuales internas)

abstract (Referencia de C#)

18/03/2020 • 5 minutes to read • Edit Online

El modificador `abstract` indica que lo que se modifica carece de implementación o tiene una implementación incompleta. El modificador abstract puede usarse con clases, métodos, propiedades, indexadores y eventos. Use el modificador `abstract` en una declaración de clase para indicar que una clase está diseñada como clase base de otras clases, no para crear instancias por sí misma. Los miembros marcados como abstractos deben implementarse con clases no abstractas derivadas de la clase abstracta.

Ejemplo

En este ejemplo, la clase `Square` debe proporcionar una implementación de `GetArea` porque se deriva de `Shape`:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Las clases abstractas tienen las siguientes características:

- No se pueden crear instancias de una clase abstracta.
- Una clase abstracta puede contener descriptores de acceso y métodos abstractos.
- No es posible modificar una clase abstracta con el modificador sealed porque los dos modificadores tienen significados opuestos. El modificador `sealed` impide que una clase se herede y el modificador `abstract` requiere que una clase se herede.
- Una clase no abstracta que derive de una clase abstracta debe incluir implementaciones reales de todos los descriptores de acceso y métodos abstractos heredados.

Use el modificador `abstract` en una declaración de método o de propiedad para indicar que el método o la propiedad no contienen implementación.

Los métodos abstractos tienen las siguientes características:

- Un método abstracto es, implícitamente, un método virtual.

- Solo se permiten declaraciones de métodos abstractos en clases abstractas.
- Dado que una declaración de método abstracto no proporciona una implementación real, no hay ningún cuerpo de método; la declaración de método finaliza simplemente con un punto y coma y no hay llaves ({ }) después de la firma. Por ejemplo:

  ```
  public abstract void MyMethod();
  ```

  La implementación la proporciona un método, override, que es miembro de una clase no abstracta.
- Es un error usar los modificadores static o virtual en una declaración de método abstracto.

Las propiedades abstractas se comportan como métodos abstractos, salvo por las diferencias en la sintaxis de declaración e invocación.

- Es un error usar el modificador `abstract` en una propiedad estática.
- Una propiedad abstracta heredada se puede invalidar en una clase derivada incluyendo una declaración de propiedad que use el modificador override.

Para obtener más información sobre las clases abstractas, vea Clases y miembros de clase abstractos y sellados .

Una clase abstracta debe proporcionar implementación para todos los miembros de interfaz.

Una clase abstracta que implemente una interfaz podría asignar los métodos de interfaz a métodos abstractos. Por ejemplo:

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

Ejemplo

En este ejemplo, la clase `DerivedClass` se deriva de una clase abstracta `BaseClass`. La clase abstracta contiene un método abstracto, `AbstractMethod`, y dos propiedades abstractas, `X` y `Y`.

```
abstract class BaseClass   // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod();   // Abstract method
    public abstract int X    { get; }
    public abstract int Y    { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

En el ejemplo anterior, si intenta crear una instancia de la clase abstracta mediante una instrucción como esta:

```
BaseClass bc = new BaseClass();   // Error
```

Se mostrará un mensaje de error en el que se indica que el compilador no puede crear una instancia de la clase abstracta "BaseClass".

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Modificadores

- virtual
- override
- Palabras clave de C#

async (Referencia de C#)

02/04/2020 • 8 minutes to read • Edit Online

Use el modificador `async` para especificar que un método, una expresión lambda o un método anónimo es asincrónico. Si usa este modificador en un método o una expresión, se hace referencia al mismo como un *método asincrónico*. En el ejemplo siguiente se define un método asincrónico denominado `ExampleMethodAsync`:

```
public async Task<int> ExampleMethodAsync()
{
    // . . . .
}
```

Si no está familiarizado con la programación asincrónica o no entiende cómo un método asincrónico usa el operador `await` para hacer el trabajo de larga duración sin bloquear el subproceso del autor de la llamada, lea la introducción de Programación asincrónica con async y await. El siguiente código se encuentra dentro de un método asincrónico y llama al método HttpClient.GetStringAsync:

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

Un método asincrónico se ejecuta sincrónicamente hasta alcanzar la primera expresión `await`, en la que se suspende el método hasta que se complete la tarea en espera. Mientras tanto, el control vuelve al llamador del método, como se muestra en el ejemplo de la sección siguiente.

Si el método que la palabra clave `async` modifica no contiene una expresión o instrucción `await`, el método se ejecuta de forma sincrónica. Una advertencia del compilador alerta de cualquier método asincrónico que no contenga instrucciones de `await`, porque esa situación podría indicar un error. Vea Advertencia del compilador (nivel 1) CS4014.

La palabra clave `async` es contextual en el sentido de que es una palabra clave cuando modifica un método, una expresión lambda o un método anónimo. En todos los demás contextos, se interpreta como identificador.

Ejemplo

En el ejemplo siguiente se muestra la estructura y el flujo de control entre un controlador de eventos asincrónicos, `StartButton_Click`, y un método asincrónico, `ExampleMethodAsync`. El resultado del método asincrónico es el número de caracteres de una página web. El código es adecuado para una aplicación Windows Presentation Foundation (WPF) o de la Tienda Windows creada en Visual Studio; vea los comentarios del código para configurar la aplicación.

Puede ejecutar este código en Visual Studio como una aplicación Windows Presentation Foundation (WPF) o una aplicación de la Tienda Windows. Necesita un control de botón denominado `StartButton` y un control de cuadro de texto denominado `ResultsTextBox`. Recuerde establecer los nombres y el controlador de manera que tenga algo similar a esto:

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top"
Width="75"
        Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"
         Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>
```

Para ejecutar el código como una aplicación WPF:

- Pegue este código en la clase `MainWindow` en MainWindow.xaml.cs.
- Agregue una referencia a System.Net.Http.
- Agregue una directiva `using` a System.Net.Http.

Para ejecutar el código como una aplicación de la Tienda Windows:

- Pegue este código en la clase `MainPage` en MainPage.xaml.cs.
- Agregue directivas using para System.Net.Http y System.Threading.Tasks.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next line where
        // "length" is, but due to when '+=' fetches the value of ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

IMPORTANT

Para más información sobre las tareas y el código que se ejecuta mientras se espera la finalización por una tarea, vea Programación asincrónica con async y await. Para obtener un ejemplo completo de WPF en el que se usan elementos similares, vea Tutorial: Acceso a la Web usando async y await.

Tipos de valor devueltos

Un método asincrónico puede tener los siguientes tipos de valor devuelto:

- Task
- Task<TResult>
- void. Los métodos `async void` no suelen ser recomendables para código que no sea controladores de eventos dado que los autores de la llamada no pueden usar `await` con esos métodos y deben implementar otro mecanismo para informar sobre la finalización correcta o condiciones de error.
- A partir de C# 7.0, cualquier tipo que tenga un método `GetAwaiter` accesible. El tipo `System.Threading.Tasks.ValueTask<TResult>` es una implementación de ese tipo. Está disponible agregando el paquete NuGet `System.Threading.Tasks.Extensions`.

El método asincrónico no puede declarar ningún parámetro in, ref o out, ni puede tener un valor devuelto de referencia, pero puede llamar a los métodos que tienen estos parámetros.

Se puede especificar `Task<TResult>` como el tipo de valor devuelto de un método asincrónico si la instrucción return del método especifica un operando de tipo `TResult`. Utilice `Task` si no se devuelve ningún valor significativo al completarse el método. Es decir, una llamada al método devuelve `Task`, pero cuando se completa `Task`, las expresiones `await` que esperan a que `Task` finalice se evalúan como `void`.

El tipo devuelto `void` se utiliza principalmente para definir controladores de eventos, que requieren ese tipo devuelto. El llamador de un método asincrónico que devuelva `void` no puede esperar a que finalice y no puede detectar las excepciones que el método inicia.

A partir de C# 7.0, devuelve otro tipo, normalmente un tipo de valor, que tiene un método `GetAwaiter` para minimizar las asignaciones de memoria en secciones críticas de rendimiento del código.

Para más información y ejemplos, vea Tipos de valor devueltos asincrónicos.

Vea también

- AsyncStateMachineAttribute
- await
- Tutorial: Acceso a la Web usando async y await
- Programación asincrónica con Async y Await

const (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

La palabra clave `const` se usa para declarar un campo constante o una local constante. Los campos y locales constantes no son variables y no se pueden modificar. Las constantes pueden ser números, valores booleanos, cadenas o una referencia nula. No cree una constante para representar información que espera que cambie en algún momento. Por ejemplo, no use un campo constante para almacenar el precio de un servicio, un número de versión de producto o el nombre comercial de una compañía. Estos valores pueden cambiar con el tiempo y, como los compiladores propagan las constantes, otro código compilado con sus bibliotecas tendrán que volver a compilarse para ver los cambios. Vea también la palabra clave readonly. Por ejemplo:

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

Comentarios

El tipo de una declaración constante especifica el tipo de los miembros que la declaración presenta. El inicializador de una local constante o de un campo constante debe ser una expresión constante que se pueda convertir implícitamente al tipo de destino.

Una expresión constante es una expresión que se puede evaluar por completo en tiempo de compilación. Por lo tanto, los únicos valores posibles para las constantes de tipos de referencia son `string` y una referencia nula.

La declaración de constante puede declarar varias constantes, tales como:

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

El modificador `static` no se permite en una declaración de constante.

Una constante puede participar en una expresión constante, de la siguiente manera:

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

> **NOTE**
>
> La palabra clave readonly difiere de la palabra clave `const`. Un campo `const` solo se puede inicializar en la declaración del campo. Un campo `readonly` se puede inicializar en la declaración o en un constructor. Por lo tanto, los campos `readonly` pueden tener diferentes valores en función del constructor que se use. Además, aunque un campo `const` es una constante en tiempo de compilación, el campo `readonly` se puede usar para constantes en tiempo de ejecución, como en esta línea: `public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

Ejemplo

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
    x = 11, y = 22
    C1 = 5, C2 = 10
*/
```

Ejemplo

Este ejemplo demuestra cómo usar las constantes como variables locales.

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores
- readonly

event (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

La palabra clave `event` se usa para declarar un evento en una clase de publicador.

Ejemplo

En el ejemplo siguiente se muestra cómo declarar y generar un evento que usa EventHandler como el tipo de delegado subyacente. Para obtener el código de ejemplo completo que también muestra cómo usar el tipo delegado EventHandler<TEventArgs> genérico y cómo suscribirse a un evento y crear un método de controlador de evento, consulte Procedimiento Publicar eventos que cumplan las directrices de .NET Framework.

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

Los eventos son un tipo especial de delegado de multidifusión que solo se pueden invocar desde la clase o el struct en la que se declaran (la clase de publicador). Si otras clases o structs se suscriben al evento, se llamará a sus métodos de controlador de eventos cuando la clase de publicador genera el evento. Para más información y ejemplos de código, vea Eventos y Delegados.

Las constantes pueden marcarse como public, private, protected, internal, protected internal o private protected. Estos modificadores de acceso definen cómo los usuarios de la clase pueden obtener acceso al evento. Para obtener más información, vea Modificadores de acceso.

Palabras clave y eventos

Las palabras clave siguientes se aplican a eventos.

PALABRA CLAVE	DESCRIPTION	PARA OBTENER MÁS INFORMACIÓN

PALABRA CLAVE	DESCRIPTION	PARA OBTENER MÁS INFORMACIÓN
static	Hace que el evento esté disponible para los llamadores en cualquier momento, aunque no exista ninguna instancia de la clase.	Clases estáticas y sus miembros
virtual	Permite que las clases derivadas invaliden el comportamiento de eventos mediante la palabra clave override.	Herencia
sealed	Especifica que ya no es virtual para las clases derivadas.	
abstract	El compilador no generará los bloques de descriptor de acceso de eventos `add` y `remove`, y por tanto, las clases deben proporcionar su propia implementación.	

Un evento puede declararse como evento estático mediante la palabra clave static. Esto hace que el evento esté disponible para los llamadores en cualquier momento, aunque no exista ninguna instancia de la clase. Para obtener más información, consulte Clases estáticas y sus miembros.

Un evento puede marcarse como virtual mediante la palabra clave virtual. Esto permite que las clases derivadas invaliden el comportamiento de eventos mediante la palabra clave override. Para obtener más información, vea Herencia. Un evento que reemplaza un evento virtual también puede ser sealed, que especifica que ya no es virtual para las clases derivadas. Por último, se puede declarar un evento como abstract, lo que significa que el compilador no generará los bloques de descriptor de acceso de eventos `add` y `remove`. Por tanto, las clases derivadas deben proporcionar una implementación propia.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- add
- remove
- Modificadores
- Procedimiento para combinar delegados (delegados de multidifusión)

extern (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

El modificador `extern` se usa para declarar un método que se implementa externamente. Un uso común del modificador `extern` es con el atributo `DllImport` al usar servicios de interoperabilidad para llamar a código no administrado. En este caso, el método se debe declarar también como `static`, como se muestra en el ejemplo siguiente:

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

La palabra clave `extern` también puede definir un alias del ensamblado externo, lo que permite hacer referencia a diferentes versiones del mismo componente desde un único ensamblado. Para obtener más información, vea alias externo.

Es un error usar los modificadores abstract y `extern` juntos para modificar el mismo miembro. El uso del modificador `extern` significa que el método se implementa fuera del código de C#, mientras que el uso del modificador `abstract` significa que la implementación del método no se proporciona en la clase.

La palabra clave extern tiene usos más limitados en C# que en C++. Para comparar la palabra clave de C# con la de C++, consulte el tema sobre el uso de extern para especificar vinculación en la referencia del lenguaje C++.

Ejemplo 1

En este ejemplo, el programa recibe una cadena del usuario y la muestra en un cuadro de mensaje. El programa usa el método `MessageBox` importado de la biblioteca User32.dll.

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

Ejemplo 2

En este ejemplo se muestra un programa de C# que llama a una biblioteca de C (una DLL nativa).

1. Cree el archivo de C siguiente y denomínelo `cmdll.c`:

```
// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
  return i*10;
}
```

2. Abra una ventana del símbolo del sistema de las herramientas nativas de Visual Studio x64 (o x32) desde el directorio de instalación de Visual Studio y compile el archivo `cmdll.c` escribiendo **cl -LD cmdll.c** en el símbolo del sistema.

3. En el mismo directorio, cree el siguiente archivo de C# y denomínelo `cm.cs`:

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
      public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}
```

4. Abra una ventana del símbolo del sistema de las herramientas nativas de Visual Studio x64 (o x32) del directorio de instalación de Visual Studio y compile el archivo `cm.cs` escribiendo:

   > **csc cm.cs** (para el símbolo del sistema x64), o bien **csc -platform:x86 cm.cs** (para el símbolo del sistema x32)

   De esta forma, se creará el archivo ejecutable `cm.exe`.

5. Ejecute `cm.exe`. El método `SampleMethod` pasa el valor 5 al archivo DLL, que devuelve el valor multiplicado por 10. El programa produce el siguiente resultado:

```
SampleMethod() returns 50.
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- System.Runtime.InteropServices.DllImportAttribute
- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores

in (Modificador genérico) (Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

Para los parámetros de tipo genérico, la palabra clave `in` especifica que el parámetro de tipo es contravariante. Puede usar la palabra clave `in` en las interfaces y delegados genéricos.

La contravarianza permite usar un tipo menos derivado que el que se especifica en el parámetro genérico. Esto permite la conversión implícita de las clases que implementan interfaces contravariantes y la conversión implícita de los tipos de delegado. La covarianza y la contravarianza de los parámetros de tipo genérico son compatibles con los tipos de referencia, pero no lo son con los tipos de valor.

Un tipo se puede declarar contravariante en una interfaz o un delgado genéricos solo si define el tipo de parámetros de un método y no el tipo de valor devuelto de un método. Los parámetros `In`, `ref` y `out` deben ser invariables, es decir, ni covariantes ni contravariantes.

Una interfaz que tiene un parámetro de tipo contravariante permite que sus métodos acepten argumentos de tipos menos derivados que los que se especifican en el parámetro de tipo de interfaz. Por ejemplo, en la interfaz IComparer<T>, el tipo T es contravariante, puede asignar un objeto de tipo `IComparer<Person>` a un objeto de tipo `IComparer<Employee>` sin tener que usar ningún método de conversión especial si `Employee` hereda `Person`.

A un delegado contravariante se le puede asignar otro delegado del mismo tipo, pero con un parámetro de tipo genérico menos derivado.

Para obtener más información, vea Covariance and Contravariance (Covarianza y contravarianza).

Interfaz genérica contravariante

En el ejemplo siguiente se muestra cómo declarar, extender e implementar una interfaz genérica contravariante. También se muestra cómo puede usar la conversión implícita para las clases que implementan esta interfaz.

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

Delegado genérico contravariante

En el ejemplo siguiente se muestra cómo declarar e invocar un delegado genérico contravariante, y crear instancias

de este. También se muestra cómo puede convertir implícitamente un tipo de delegado.

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- out
- Covarianza y contravarianza
- Modificadores

new (Modificador, Referencia de C#)

29/06/2020 • 6 minutes to read • Edit Online

Cuando se utiliza como modificador de una declaración, la palabra clave `new` oculta explícitamente un miembro heredado de una clase base. Cuando se oculta un miembro heredado, la versión derivada del miembro reemplaza a la versión de la clase base. Aunque los miembros se pueden ocultar sin utilizar el modificador `new`, obtendrá una advertencia del compilador. Si utiliza `new` explícitamente para ocultar un miembro, se suprime esta advertencia.

También puede usar la palabra clave `new` para crear una instancia de un tipo o como una restricción de tipo genérico.

Para ocultar un miembro heredado, declárelo en la clase derivada con el mismo nombre de miembro y modifíquelo con la palabra clave `new`. Por ejemplo:

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

En este ejemplo, `BaseC.Invoke` oculta `DerivedC.Invoke`. El campo `x` no se ve afectado porque no lo oculta un nombre similar.

La ocultación de nombres por medio de la herencia toma una de las siguientes formas:

- Normalmente, una constante, un campo, una propiedad o un tipo que se muestran en una clase o struct ocultan a todos los miembros de la clase base que comparten su nombre. Hay casos especiales. Por ejemplo, si declara un nuevo campo con el nombre `N` para tener un tipo que no es invocable y un tipo base declara `N` como método, el nuevo campo no oculta la declaración base en la sintaxis de invocación. Para obtener más información, vea la sección Búsqueda de miembros de la Especificación del lenguaje C#.
- Un método introducido en una clase o struct oculta las propiedades, los campos y los tipos que comparten el nombre en la clase base. También oculta todos los métodos de la clase base que tienen la misma signatura.
- Un indizador introducido en una clase o struct oculta todos los indizadores de la clase base que tienen la misma signatura.

Es un error usar `new` y override en el mismo miembro, porque los dos modificadores tienen significados mutuamente excluyentes. El modificador `new` crea un nuevo miembro con el mismo nombre y oculta el miembro original. El modificador `override` amplía la implementación de un miembro heredado.

Si se utiliza el modificador `new` en una declaración que no oculta un miembro heredado, se genera una advertencia.

Ejemplo

En este ejemplo, una clase base, `BaseC`, y una clase derivada, `DerivedC`, utilizan el mismo nombre de campo `x`, lo que oculta el valor del campo heredado. El ejemplo muestra el uso del modificador `new`. También muestra cómo obtener acceso a los miembros ocultos de la clase base mediante sus nombres completos.

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

Ejemplo

En este ejemplo, una clase anidada oculta una clase que tiene el mismo nombre en la clase base. El ejemplo muestra cómo utilizar el modificador `new` para eliminar el mensaje de advertencia y cómo obtener acceso a los miembros de la clase oculta mediante sus nombres completos.

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1  = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

Si quita el modificador `new`, el programa seguirá compilándose y ejecutándose, pero aparecerá la siguiente advertencia:

```
The keyword new is required on 'MyDerivedC.x' because it hides inherited member 'MyBaseC.x'.
```

Especificación del lenguaje C#

Para obtener más información, vea la sección El modificador new de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores
- Control de versiones con las palabras clave Override y New
- Saber cuándo utilizar las palabras clave Override y New

out (Modificador genérico) (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

Para los parámetros de tipo genérico, la palabra clave `out` especifica que el parámetro de tipo es covariante. Puede usar la palabra clave `out` en las interfaces y delegados genéricos.

La covarianza le permite usar un tipo más derivado que el que se especifica en el parámetro genérico. Esto permite la conversión implícita de las clases que implementan interfaces covariantes y la conversión implícita de los tipos de delegado. La covarianza y la contravarianza son compatibles con los tipos de referencia, pero no lo son con los tipos de valor.

Una interfaz con un parámetro de tipo covariante permite que sus métodos devuelvan tipos más derivados que los especificados por el parámetro de tipo. Por ejemplo, dado que en .NET Framework 4, en IEnumerable<T>, el tipo T es covariante, puede asignar un objeto del tipo `IEnumerable(Of String)` a otro objeto del tipo `IEnumerable(Of Object)` sin usar ningún método de conversión especial.

A un delegado covariante se le puede asignar otro delegado del mismo tipo, pero con un parámetro de tipo genérico más derivado.

Para obtener más información, vea Covariance and Contravariance (Covarianza y contravarianza).

Ejemplo: interfaz genérica covariante

En el ejemplo siguiente se muestra cómo declarar, extender e implementar una interfaz genérica covariante. También se muestra cómo usar la conversión implícita para las clases que implementan una interfaz covariante.

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}
```

En una interfaz genérica, un parámetro de tipo se puede declarar como covariante si cumple las siguientes condiciones:

- El parámetro de tipo se usa solamente como tipo de valor devuelto de los métodos de interfaz y no como tipo de los argumentos de método.

> **NOTE**
> Hay una excepción para esta regla. Si en una interfaz covariante tiene un delegado genérico contravariante como parámetro de método, puede usar el tipo covariante como parámetro de tipo genérico para este delegado. Para obtener más información sobre los delegados genéricos covariantes y contravariantes, vea Varianza en delegados y Usar la varianza para los delegados genéricos Func y Action.

- El parámetro de tipo no se usa como restricción genérica para los métodos de interfaz.

Ejemplo: delegado genérico covariante

En el ejemplo siguiente se muestra cómo declarar, invocar y crear instancias de un delegado genérico covariante. También se muestra cómo convertir implícitamente los tipos de delegado.

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

En un delegado genérico, un tipo se puede declarar como covariante si se usa solamente como tipo de valor devuelto por un método y no se usa para los argumentos de método.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Varianza en interfaces genéricas
- in
- Modificadores

override (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

El modificador `override` es necesario para ampliar o modificar la implementación abstracta o virtual de un método, propiedad, indexador o evento heredado.

Ejemplo

En este ejemplo, la clase `Square` debe proporcionar una implementación de invalidación de `GetArea` porque `GetArea` se hereda de la clase abstracta `Shape`:

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Un método `override` proporciona una nueva implementación de un miembro que se hereda de una clase base. El método invalidado por una declaración `override` se conoce como método base invalidado. El método base invalidado debe tener la misma firma que el método `override`. Para obtener información sobre la herencia, vea Herencia.

No se puede invalidar un método estático o no virtual. El método base invalidado debe ser `virtual`, `abstract` o `override`.

Una declaración `override` no puede cambiar la accesibilidad del método `virtual`. El método `override` y el método `virtual` deben tener el mismo modificador de nivel de acceso.

No se pueden usar los modificadores `new`, `static` o `virtual` para modificar un método `override`.

Una declaración de propiedad de invalidación debe especificar exactamente el mismo modificador de acceso, tipo y nombre que la propiedad heredada, y la propiedad invalidada debe ser `virtual`, `abstract` o `override`.

Para obtener más información sobre cómo usar la palabra clave `override`, vea Control de versiones con las palabras clave Override y New y Saber cuándo usar las palabras clave Override y New (Guía de programación de C#).

Ejemplo

En este ejemplo se define una clase base denominada `Employee` y una clase derivada denominada `SalesEmployee`. La clase `SalesEmployee` incluye un campo adicional, `salesbonus`, e invalida el método `CalculatePay` para tenerlo en cuenta.

```
class TestOverride
{
    public class Employee
    {
        public string name;

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay,
                  decimal salesbonus) : base(name, basepay)
        {
            this.salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return basepay + salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice",
                      1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine("Employee4 " + employee1.name +
                  " earned: " + employee1.CalculatePay());
        Console.WriteLine("Employee4 " + employee2.name +
                  " earned: " + employee2.CalculatePay());
    }
}
/*
    Output:
    Employee4 Alice earned: 1500
    Employee4 Bob earned: 1200
*/
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Herencia
- Palabras clave de C#
- Modificadores
- abstract
- virtual
- new (modificador)
- Polimorfismo

readonly (Referencia de C#)

29/06/2020 • 7 minutes to read • Edit Online

La palabra clave `readonly` es un modificador que se puede usar en cuatro contextos:

- En una declaración de campo, `readonly` indica que la asignación a un campo solo se puede producir como parte de la declaración o en un constructor de la misma clase. Se puede asignar y reasignar varias veces un campo de solo lectura dentro de la declaración de campo y el constructor.

  No se puede asignar un campo `readonly` después de que el constructor salga. Esta regla tiene diferentes implicaciones para los tipos de valor y tipos de referencia:

  - Debido a que los tipos de valor contienen directamente sus datos, un campo que es un tipo de valor `readonly` es inmutable.
  - Dado que los tipos de referencia contienen una referencia a sus datos, un campo que es un tipo de referencia `readonly` debe referirse siempre al mismo objeto. Ese objeto no es inmutable. El modificador `readonly` evita que el campo se reemplace por una instancia diferente del tipo de referencia. Sin embargo, el modificador no impide que los datos de instancia del campo se modifiquen a través del campo de solo lectura.

  > **WARNING**
  >
  > Un tipo visible externamente que contenga un campo de solo lectura visible externamente que sea un tipo de referencia mutable puede ser una vulnerabilidad de seguridad y puede desencadenar la advertencia CA2104: "No declarar tipos de referencias mutables de solo lectura".

- En una definición de tipo de `readonly struct`, `readonly` indica que el tipo de estructura es inmutable. Para obtener más información, vea la sección struct `readonly` del artículo tipos de estructura.

- En una declaración de miembro de instancia dentro de un tipo de estructura, `readonly` indica que un miembro de instancia no modifica el estado de la estructura. Para obtener más información, vea la sección Miembros de instancia `readonly` del artículo Tipos de estructura.

- En una devolución del método `ref readonly`, el modificador `readonly` indica que el método devuelve una referencia y las operaciones de escritura no se permiten en esa referencia.

Los contextos `readonly struct` y `ref readonly` se han agregado en C# 7.2. Los miembros de estructura `readonly` se han agregado en C# 8.0.

Ejemplo de campo readonly

En este ejemplo, el valor del campo `year` no se puede cambiar en el método `ChangeYear`, aunque se asigne un valor en el constructor de clase:

```
class Age
{
    readonly int year;
    Age(int year)
    {
        this.year = year;
    }
    void ChangeYear()
    {
        //year = 1967; // Compile error if uncommented.
    }
}
```

Solo se puede asignar un valor a un campo `readonly` en los siguientes contextos:

- Cuando la variable se inicializa en la declaración, por ejemplo:

  ```
  public readonly int y = 5;
  ```

- En un constructor de instancia de la clase que contiene la declaración de campo de instancia.
- En el constructor estático de la clase que contiene la declaración de campo estático.

Estos contextos de constructor son también los únicos en los que es válido pasar un campo `readonly` como parámetro out o ref.

NOTE

La palabra clave `readonly` es diferente de la palabra clave const. Un campo `const` solo se puede inicializar en la declaración del campo. Un campo `readonly` se puede asignar varias veces en la declaración de campo y en cualquier constructor. Por lo tanto, los campos `readonly` pueden tener diferentes valores en función del constructor que se use. Además, mientras que un campo `const` es una constante en tiempo de compilación, el campo `readonly` puede usarse para constantes en tiempo de ejecución, como muestra el siguiente ejemplo:

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);   // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;   // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
     Output:
        p1: x=11, y=21, z=32
        p2: x=55, y=25, z=24
    */
}
```

En el ejemplo anterior, si se usa una instrucción como el ejemplo siguiente:

```
p2.y = 66;        // Error
```

se obtendrá el siguiente mensaje de error del compilador:

No se puede asignar un campo de solo lectura (excepto en un constructor o inicializador de variable) .

Ejemplo de devolución de Ref readonly

El modificador `readonly` en un elemento `ref return` indica que la referencia devuelta no se puede modificar. En el ejemplo siguiente se devuelve una referencia al origen. Usa el modificador `readonly` para indicar que los autores de la llamada no pueden modificar el origen:

```
private static readonly SamplePoint origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref origin;
```

No es necesario que el tipo devuelto sea una `readonly struct`. Cualquier tipo que pueda devolver `ref` también puede devolver `ref readonly`.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la

fuente definitiva de la sintaxis y el uso de C#.

También puede ver las propuestas de especificación del lenguaje:

- referencia de solo lectura y estructura de solo lectura
- miembros de estructura de solo lectura

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores
- const
- Campos

sealed (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

Cuando se aplica a una clase, el modificador `sealed` impide que otras clases hereden de ella. En el ejemplo siguiente, la clase `B` hereda de la clase `A`, pero ninguna clase puede heredar de la clase `B`.

```
class A {}
sealed class B : A {}
```

También puede usar el modificador `sealed` en un método o propiedad que invalida un método o propiedad virtual en una clase base. De este modo, puede permitir que las clases deriven de su clase e impedir que invaliden determinados métodos o propiedades virtuales.

Ejemplo

En el ejemplo siguiente, `Z` hereda de `Y` pero `Z` no puede invalidar la función virtual `F` que se declara en `X` y se sella en `Y`.

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Al definir nuevos métodos o propiedades en una clase, puede impedir que las clases derivadas los invaliden. Para ello, no los declare como virtuales.

Es un error usar el modificador abstract con una clase sellada, porque una clase abstracta debe heredarla una clase que proporcione una implementación de los métodos o propiedades abstractos.

Cuando se aplica a un método o propiedad, el modificador `sealed` siempre se debe usar con override.

Dado que los structs están sellados implícitamente, no puede heredarse.

Para obtener más información, vea Herencia.

Para obtener más ejemplos, vea Clases y miembros de clase abstractos y sellados.

Ejemplo

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

En el ejemplo anterior, podría intentar heredar de la clase sellada mediante la instrucción siguiente:

```
class MyDerivedC: SealedClass {} // Error
```

El resultado es un mensaje de error:

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

Comentarios

Para determinar si se debe sellar una clase, un método o una propiedad, por lo general debe tener en cuenta los dos puntos siguientes:

- Las posibles ventajas que podrían obtener las clases derivadas con la capacidad de personalizar la clase.
- La posibilidad de que las clases derivadas modifiquen las clases de tal manera que no funcionen correctamente o del modo esperado.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Clases estáticas y sus miembros
- Clases y miembros de clase abstractos y sellados
- Modificadores de acceso
- Modificadores
- override
- virtual

static (Referencia de C#)

04/05/2020 • 5 minutes to read • Edit Online

En esta página se trata la palabra clave del modificador `static`. La palabra clave `static` también forma parte de la directiva `using static`.

Use el modificador `static` para declarar un miembro estático, que pertenece al propio tipo en lugar de a un objeto específico. El modificador `static` se puede usar para declarar clases `static`. En las clases, las interfaces y las estructuras, puede agregar el modificador `static` a los campos, los métodos, las propiedades, los operadores, los eventos y los constructores. El modificador `static` no se puede usar con indizadores ni finalizadores. Para más información, vea Clases estáticas y sus miembros.

Ejemplo: clase estática

La siguiente clase se declara como `static` y contiene solo métodos `static`:

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}
```

Una declaración de constantes o tipos es implícitamente un miembro `static`. No se puede hacer referencia a un miembro `static` mediante una instancia, sino que se hace a través del nombre de tipo. Por ejemplo, considere la siguiente clase:

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

Para hacer referencia al miembro `static` `x`, use el nombre completo, `MyBaseC.MyStruct.x`, a menos que el miembro sea accesible desde el mismo ámbito:

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

Mientras que una instancia de una clase contiene una copia independiente de todos los campos de instancia de la clase, solo hay una copia de cada campo `static`.

No se puede usar `this` para hacer referencia a métodos `static` o descriptores de acceso de propiedades.

Si la palabra clave `static` se aplica a una clase, todos los miembros de esta deben ser `static`.

Las clases, las interfaces y las clases `static` pueden tener constructores `static`. Se llama a un constructor `static` en algún momento entre el inicio del programa y la creación de una instancia de la clase.

> **NOTE**
> La palabra clave `static` tiene usos más limitados que en C++. Para ver una comparación con la palabra clave de C++, vea Clases de almacenamiento (C++).

Para mostrar miembros `static`, es recomendable una clase que represente al empleado de una empresa. Supongamos que la clase contiene un método de recuento de empleados y un campo para almacenar el número de empleados. El método y el campo no pertenecen a ninguna instancia de ningún empleado, sino que pertenecen a la clase de empleados en su conjunto. Se deben declarar como miembros `static` de la clase.

Ejemplo: campo estático y método

En este ejemplo se lee el nombre y el identificador de un nuevo empleado, se incrementa en uno el recuento de empleados y se muestra la información del nuevo empleado, así como el nuevo número de empleados. Este programa lee el número actual de empleados desde el teclado.

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID:   {e.id}");
        Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
 *
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID:   AF643G
New Number of Employees: 16
*/
```

Ejemplo: inicialización estática

En este ejemplo se muestra que se puede inicializar un campo `static` mediante otro campo `static` que aún no se ha declarado. Los resultados están sin definir hasta que se asigna explícitamente un valor al campo `static`.

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
    0
    5
    99
*/
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores
- using static (directiva)
- Clases estáticas y sus miembros

unsafe (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `unsafe` denota un contexto no seguro, que es necesario para realizar cualquier operación que implique punteros. Para obtener más información, vea Código no seguro y punteros (Guía de programación de C#).

Puede usar el codificador `unsafe` en la declaración de un tipo o miembro. Por consiguiente, toda la extensión textual del tipo o miembro se considera un contexto no seguro. Por ejemplo, el siguiente método se declara con el modificador `unsafe`:

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

El ámbito del contexto no seguro se extiende desde la lista de parámetros hasta el final del método, por lo que también pueden usarse punteros en la lista de parámetros:

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

También puede usarse un bloque no seguro para habilitar el uso de código no seguro en el bloque. Por ejemplo:

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

Para compilar código no seguro, debe especificar la opción del compilador `-unsafe`. Common Language Runtime no puede comprobar el código no seguro.

Ejemplo

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

especificación del lenguaje C#

Para más información, vea la sección sobre código no seguro de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- fixed (instrucción)
- Código no seguro y punteros
- Búferes de tamaño fijo

virtual (Referencia de C#)

18/03/2020 • 5 minutes to read • Edit Online

La palabra clave `virtual` se usa para modificar una declaración de método, propiedad, indizador o evento y permitir que se invalide en una clase derivada. Por ejemplo, cualquier clase que herede este método puede reemplazarlo:

```
public virtual double Area()
{
    return x * y;
}
```

Un miembro de reemplazo de una clase derivada puede modificar la implementación de un miembro virtual. Para obtener más información sobre cómo usar la palabra clave `virtual`, vea Control de versiones con las palabras clave Override y New y Saber cuándo usar las palabras clave Override y New (Guía de programación de C#).

Comentarios

Cuando se invoca a un método virtual, se busca un miembro de reemplazo en el tipo en tiempo de ejecución del objeto. Se llama al miembro de reemplazo en la clase más derivada, que podría ser el miembro original si ninguna clase derivada ha invalidado al miembro.

De forma predeterminada, los métodos son no virtuales. No se puede invalidar un método no virtual.

El modificador `virtual` no se puede usar con los modificadores `static`, `abstract`, `private` o `override`. En el siguiente ejemplo se muestra una propiedad virtual:

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
```

Las propiedades virtuales se comportan como métodos virtuales, salvo por las diferencias en la sintaxis de declaración e invocación.

- Es un error usar el modificador `virtual` en una propiedad estática.
- Una propiedad virtual heredada se puede invalidar en una clase derivada al incluir una declaración de propiedad que use el modificador `override`.

Ejemplo

En este ejemplo, la clase `Shape` contiene las dos coordenadas `x`, `y` y el método virtual `Area()`. Las distintas clases de formas como `Circle`, `Cylinder` y `Sphere` heredan la clase `Shape` y se calcula el área de cada figura. Cada clase derivada tiene su propia implementación de invalidación de `Area()`.

Observe que las clases heredadas `Circle`, `Sphere` y `Cylinder` usan constructores que inicializan la clase base, como se muestra en la siguiente declaración.

```
public Cylinder(double r, double h): base(r, h) {}
```

El programa siguiente calcula y muestra el área apropiada de cada figura al invocar a la implementación

adecuada del método `Area()`, según el objeto asociado al método.

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double x, y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public virtual double Area()
        {
            return x * y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * x * x;
        }
    }

    class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return 4 * PI * x * x;
        }
    }

    class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * x * x + 2 * PI * x * y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
```

```
        Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Polimorfismo
- abstract
- override
- new (modificador)

volatile (Referencia de C#)

18/03/2020 • 5 minutes to read • Edit Online

La palabra clave `volatile` indica que un campo puede ser modificado por varios subprocesos que se ejecutan al mismo tiempo. El compilador, el sistema de runtime e incluso el hardware pueden reorganizar las lecturas y escrituras en las ubicaciones de memoria por motivos de rendimiento. Los campos que se declaran `volatile` no están sujetos a estas optimizaciones. La incorporación del modificador `volatile` asegura que todos los subprocesos observarán las operaciones de escritura volátiles realizadas por cualquier otro subproceso en el orden en que se realizaron. No hay ninguna garantía de una única ordenación total de las operaciones de escritura volátiles como se muestra en todos los subprocesos de ejecución.

La palabra clave `volatile` se puede aplicar a campos de estos tipos:

- Tipos de referencia.
- Tipos de puntero (en un contexto no seguro). Observe que aunque el propio puntero puede ser volatile, no el objeto al que apunta. Es decir, no puede declarar un "puntero a volatile".
- Tipos simples como `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float` y `bool`.
- Un tipo `enum` con uno de los siguientes tipos base: `byte`, `sbyte`, `short`, `ushort`, `int` o `uint`.
- Parámetros de tipo genérico que se sabe que son tipos de referencia.
- IntPtr y UIntPtr.

Otros tipos, incluidos `double` y `long`, no pueden marcarse como `volatile` porque no se puede garantizar que las lecturas y escrituras los campos de esos tipos sean atómicas. Para proteger el acceso multiproceso a esos tipos de campos, use los miembros de clase Interlocked o proteja el acceso mediante la instrucción `lock`.

La palabra clave `volatile` solo se puede aplicar a campos de `class` o `struct`. Las variables locales no se pueden declarar como `volatile`.

Ejemplo

En el ejemplo siguiente se muestra cómo declarar una variable de campo pública como `volatile`.

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int _i)
    {
        sharedStorage = _i;
    }
}
```

En el ejemplo siguiente se muestra cómo crear un subproceso auxiliar o de trabajo y usarlo para realizar el procesamiento en paralelo con el del subproceso principal. Para más información sobre el multithreading, vea Subprocesamiento administrado.

```
public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}
```

Con el modificador `volatile` que se agrega a la declaración de `_shouldStop` en su lugar, siempre obtendrá los mismos resultados (similar al fragmento que se muestra en el código anterior). Sin embargo, sin ese modificador en el miembro `_shouldStop`, el comportamiento es imprevisible. El método `DoWork` puede optimizar el acceso a los miembros, lo que provoca la lectura de datos obsoletos. Dada la naturaleza de la programación multiproceso, el número de lecturas obsoletas es imprevisible. Distintas ejecuciones del programa generarán resultados ligeramente diferentes.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Especificación del lenguaje C#: palabra clave volatile
- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Modificadores
- lock (Instrucción)
- Interlocked

Palabras clave de instrucciones (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Las instrucciones son instrucciones de programa. A excepción de como se describe en los temas a los que se hace referencia en la tabla siguiente, las instrucciones se ejecutan en secuencia. En la tabla siguiente, se enumeran las palabras claves de instrucción de C#. Para obtener más información sobre las instrucciones que no se expresan con ninguna palabra clave, consulte Instrucciones.

CATEGORÍA	PALABRAS CLAVE DE C#
Instrucciones de selección	if, else, switch, case
Instrucciones de iteración	do, for, foreach, in, while
Instrucciones de salto	break, continue, default, goto, return, yield
Instrucciones para el control de excepciones	throw, try-catch, try-finally, try-catch-finally
Checked y unchecked	checked, unchecked
Instrucción fixed	fixed
lock (Instrucción)	lock

Vea también

- Referencia de C#
- Instrucciones
- Palabras clave de C#

if-else (Referencia de C#)

04/05/2020 • 8 minutes to read • Edit Online

Una instrucción `if` identifica qué instrucción se debe ejecutar dependiendo del valor de una expresión booleana. En el ejemplo siguiente, la variable `bool``condition` se establece en `true` y, a continuación, se comprueba en la instrucción `if` . El resultado es `The variable is set to true.`

```
bool condition = true;

if (condition)
{
    Console.WriteLine("The variable is set to true.");
}
else
{
    Console.WriteLine("The variable is set to false.");
}
```

Puede ejecutar los ejemplos de este tema situándolos en el método `Main` de una aplicación de consola.

Una instrucción `if` en C# puede tener dos formas, tal como se muestra en el ejemplo siguiente.

```
// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.

// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

En una instrucción `if-else` , si `condition` se evalúa como true, se ejecuta `then-statement` . Si `condition` es false, se ejecuta `else-statement` . Puesto que `condition` no puede ser simultáneamente true y false, la `then-statement` y la `else-statement` de una instrucción `if-else` nunca se podrán ejecutar a la vez. Después de ejecutar la `then-statement` o la `else-statement` , el control se transfiere a la siguiente instrucción situada después de la instrucción `if` .

En una instrucción `if` que no incluya una instrucción `else` , si `condition` es true, se ejecutará la `then-statement` . Si `condition` es false, el control se transfiere a la siguiente instrucción situada después de la instrucción `if` .

Tanto la `then-statement` como la `else-statement` pueden constar de una única instrucción o de varias instrucciones entre llaves (`{}`). Para una única instrucción, las llaves son opcionales pero se recomiendan.

La instrucción o las instrucciones en la `then-statement` y la `else-statement` pueden ser de cualquier tipo, incluida otra instrucción `if` anidada dentro de la instrucción `if` . En instrucciones `if` anidadas, cada

cláusula `else` pertenece a la última `if` que no tiene su `else` correspondiente. En el ejemplo siguiente, `Result1` aparece si `m > 10` y `n > 20` se evalúan como true. Si `m > 10` es true, pero `n > 20` es false, aparece `Result2` .

```
// Try with m = 12 and then with m = 8.
int m = 12;
int n = 18;

if (m > 10)
    if (n > 20)
    {
        Console.WriteLine("Result1");
    }
    else
    {
        Console.WriteLine("Result2");
    }
```

En su lugar, si desea que `Result2` aparezca cuando `(m > 10)` es false, puede especificar dicha asociación mediante llaves para establecer el inicio y el fin de la instrucción `if` anidada, como se muestra en el ejemplo siguiente.

```
// Try with m = 12 and then with m = 8.
if (m > 10)
{
    if (n > 20)
        Console.WriteLine("Result1");
}
else
{
    Console.WriteLine("Result2");
}
```

`Result2` aparece si la condición `(m > 10)` se evalúa como false.

Ejemplo

En el ejemplo siguiente, se escribe un carácter desde el teclado y el programa usa una instrucción `if` anidada para determinar si el carácter de entrada es un carácter alfabético. Si el carácter de entrada es un carácter alfabético, el programa comprueba si el carácter de entrada está en mayúsculas o minúsculas. Aparece un mensaje para cada caso.

```
Console.Write("Enter a character: ");
char c = (char)Console.Read();
if (Char.IsLetter(c))
{
    if (Char.IsLower(c))
    {
        Console.WriteLine("The character is lowercase.");
    }
    else
    {
        Console.WriteLine("The character is uppercase.");
    }
}
else
{
    Console.WriteLine("The character isn't an alphabetic character.");
}

//Sample Output:

//Enter a character: 2
//The character isn't an alphabetic character.

//Enter a character: A
//The character is uppercase.

//Enter a character: h
//The character is lowercase.
```

Ejemplo

También puede anidar una instrucción `if` dentro de un bloque else, tal como se muestra en el siguiente código parcial. El ejemplo anida instrucciones `if` dentro de dos bloques más y, a continuación, un bloque. Los comentarios de especifican qué condiciones son true o false en cada bloque.

```
// Change the values of these variables to test the results.
bool Condition1 = true;
bool Condition2 = true;
bool Condition3 = true;
bool Condition4 = true;

if (Condition1)
{
    // Condition1 is true.
}
else if (Condition2)
{
    // Condition1 is false and Condition2 is true.
}
else if (Condition3)
{
    if (Condition4)
    {
        // Condition1 and Condition2 are false. Condition3 and Condition4 are true.
    }
    else
    {
        // Condition1, Condition2, and Condition4 are false. Condition3 is true.
    }
}
else
{
    // Condition1, Condition2, and Condition3 are false.
}
```

Ejemplo

El ejemplo siguiente determina si un carácter de entrada es una letra minúscula, una letra mayúscula o un número. Si estas tres condiciones son false, el carácter no es un carácter alfanumérico. En el ejemplo se muestra un mensaje para cada caso.

```
Console.Write("Enter a character: ");
char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}

//Sample Input and Output:
//Enter a character: E
//The character is an uppercase letter.

//Enter a character: e
//The character is a lowercase letter.

//Enter a character: 4
//The character is a number.

//Enter a character: =
//The character is not alphanumeric.
```

Puesto que puede ser válida tanto una instrucción del bloque else como del bloque then, puede usar cualquier expresión booleana válida para la condición. Puede usar operadores lógicos como `!`, `&&`, `||`, `&`, `|` y `^` para hacer condiciones compuestas. En el código siguiente, se muestran algunos ejemplos:

```
// NOT
bool result = true;
if (!result)
{
    Console.WriteLine("The condition is true (result is false).");
}
else
{
    Console.WriteLine("The condition is false (result is true).");
}

// Short-circuit AND
int m = 9;
int n = 7;
int p = 5;
if (m >= n && m >= p)
{
    Console.WriteLine("Nothing is larger than m.");
}

// AND and NOT
if (m >= n && !(p > m))
{
    Console.WriteLine("Nothing is larger than m.");
}

// Short-circuit OR
if (m > n || m > p)
{
    Console.WriteLine("m isn't the smallest.");
}

// NOT and OR
m = 4;
if (!(m >= n || m >= p))
{
    Console.WriteLine("Now m is the smallest.");
}
// Output:
// The condition is false (result is true).
// Nothing is larger than m.
// Nothing is larger than m.
// m isn't the smallest.
// Now m is the smallest.
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- ?: !
- if-else (Instrucción) (C++)
- switch

switch (referencia de C#)

29/06/2020 • 26 minutes to read • Edit Online

En este artículo se trata la instrucción `switch`: Para información sobre la expresión `switch` (introducida en C# 8.0), consulte el artículo sobre expresiones `switch` en la sección expresiones y operadores.

`switch` es una instrucción de selección que elige una sola *sección switch* para ejecutarla desde una lista de candidatos en función de una coincidencia de patrones con la *expresión de coincidencia*.

```
using System;

public class Example
{
   public static void Main()
   {
      int caseSwitch = 1;

      switch (caseSwitch)
      {
          case 1:
              Console.WriteLine("Case 1");
              break;
          case 2:
              Console.WriteLine("Case 2");
              break;
          default:
              Console.WriteLine("Default case");
              break;
      }
   }
}
// The example displays the following output:
//       Case 1
```

La instrucción `switch` se suele usar como alternativa a un constructor if-else si una sola expresión se prueba con tres o más condiciones. Por ejemplo, la siguiente instrucción `switch` determina si una variable de tipo `Color` tiene uno de tres valores:

```
using System;

public enum Color { Red, Green, Blue }

public class Example
{
   public static void Main()
   {
      Color c = (Color) (new Random()).Next(0, 3);
      switch (c)
      {
         case Color.Red:
            Console.WriteLine("The color is red");
            break;
         case Color.Green:
            Console.WriteLine("The color is green");
            break;
         case Color.Blue:
            Console.WriteLine("The color is blue");
            break;
         default:
            Console.WriteLine("The color is unknown.");
            break;
      }
   }
}
```

Es equivalente al siguiente ejemplo que usa un constructor `if` - `else`.

```
using System;

public enum Color { Red, Green, Blue }

public class Example
{
   public static void Main()
   {
      Color c = (Color) (new Random()).Next(0, 3);
      if (c == Color.Red)
         Console.WriteLine("The color is red");
      else if (c == Color.Green)
         Console.WriteLine("The color is green");
      else if (c == Color.Blue)
         Console.WriteLine("The color is blue");
      else
         Console.WriteLine("The color is unknown.");
   }
}
// The example displays the following output:
//       The color is red
```

Expresión de coincidencia

La expresión de coincidencia proporciona el valor que debe coincidir con los patrones de las etiquetas `case`. Su sintaxis es:

```
   switch (expr)
```

En C# 6 y versiones anteriores, la expresión de coincidencia debe ser una expresión que devuelva un valor de los siguientes tipos:

- Un carácter.
- Una cadena.
- Un booleano.
- Un valor entero, como `int` o `long`.
- Un valor enum.

A partir de C# 7.0, la expresión de coincidencia puede ser cualquier expresión que no sea nula.

Sección switch

Una instrucción `switch` incluye una o más secciones switch. Cada sección switch contiene una o más *etiquetas case* (ya sea una etiqueta case o default) seguidas de una o más instrucciones. La instrucción `switch` puede incluir como máximo una etiqueta default colocada en cualquier sección switch. En el ejemplo siguiente se muestra una instrucción `switch` simple con tres secciones switch, cada una de ellas contiene dos instrucciones. La segunda sección switch contiene las etiquetas `case 2:` y `case 3:`.

Una instrucción `switch` puede incluir cualquier número de secciones switch y cada sección puede tener una o más etiquetas case, como se muestra en el ejemplo siguiente. Pero dos etiquetas case no pueden contener la misma expresión.

```
using System;

public class Example
{
   public static void Main()
   {
      Random rnd = new Random();
      int caseSwitch = rnd.Next(1,4);

      switch (caseSwitch)
      {
          case 1:
              Console.WriteLine("Case 1");
              break;
          case 2:
          case 3:
              Console.WriteLine($"Case {caseSwitch}");
              break;
          default:
              Console.WriteLine($"An unexpected value ({caseSwitch})");
              break;
      }
   }
}
// The example displays output like the following:
//       Case 1
```

Solo se ejecuta una sección switch en una instrucción switch. C# no permite que la ejecución continúe de una sección switch a la siguiente. Por eso, el código siguiente genera un error del compilador, CS0163: "El control no puede pasar explícitamente de una etiqueta case a otra (<case label>)".

```
switch (caseSwitch)
{
    // The following switch section causes an error.
    case 1:
        Console.WriteLine("Case 1...");
        // Add a break or other jump statement here.
    case 2:
        Console.WriteLine("... and/or Case 2");
        break;
}
```

Este requisito se suele cumplir al salir explícitamente de la sección switch mediante una instrucción break, goto o return. Pero el código siguiente también es válido, porque garantiza que el control del programa no puede pasar explícitamente a la sección switch `default`.

```
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1...");
        break;
    case 2:
    case 3:
        Console.WriteLine("... and/or Case 2");
        break;
    case 4:
        while (true)
           Console.WriteLine("Endless looping. . . .");
    default:
        Console.WriteLine("Default value...");
        break;
}
```

La ejecución de la lista de instrucciones en la sección switch con una etiqueta case que coincide con la expresión de coincidencia comienza con la primera instrucción y continúa a lo largo de la lista de instrucciones, normalmente hasta que se alcanza una instrucción de salto, como `break`, `goto case`, `goto label`, `return` o `throw`. En este punto, el control se transfiere fuera de la instrucción `switch` o a otra etiqueta case. Una instrucción `goto`, si se usa, debe transferir el control a una etiqueta de constante. Esta restricción es necesaria, ya que el intento de transferir el control a una etiqueta que no es de constante puede tener efectos secundarios no deseados, como la transferencia de control a una ubicación no deseada en el código o la creación de un bucle sin fin.

Etiquetas case

Cada etiqueta case especifica un patrón que se compara con la expresión de coincidencia (la variable `caseSwitch` en los ejemplos anteriores). Si coinciden, el control se transfiere a la sección switch que contiene la **primera** etiqueta case coincidente. Si ningún patrón de etiqueta case coincide con la expresión de coincidencia, el control se transfiere a la sección con la etiqueta case `default`, si la hubiera. Si no hay ninguna etiqueta case `default`, no se ejecuta ninguna instrucción de ninguna sección switch y el control se transfiere fuera de la instrucción `switch`.

Para más información sobre la instrucción `switch` y la coincidencia de patrones, vea la sección [Coincidencia de patrones con la instrucción `switch`](#pattern-matching with-the-switch-statement).

Dado que C# 6 solo admite el patrón constante y no permite la repetición de valores constantes, las etiquetas case definen valores mutuamente exclusivos y solo un patrón puede coincidir con la expresión de coincidencia. Por este motivo, el orden en que aparezcan las instrucciones `case` no tiene importancia.

Pero en C# 7.0, dado que se admiten otros patrones, las etiquetas de caso no necesitan definir valores

mutuamente exclusivos y varios patrones pueden coincidir con la expresión de coincidencia. Puesto que solo se ejecutan las instrucciones de la primera sección switch que contiene el patrón coincidente, el orden en que aparecen las instrucciones `case` sí es importante. Si C# detecta una sección switch cuya instrucción o instrucciones case son equivalentes a o son subconjuntos de instrucciones anteriores, genera un error del compilador, CS8120: "El caso del modificador ya se ha gestionado en un caso anterior".

En el ejemplo siguiente se muestra una instrucción `switch` que usa una variedad de patrones que no son mutuamente excluyentes. Si mueve la sección switch `case 0:` de modo que ya no sea la primera sección de la instrucción `switch`, C# genera un error del compilador debido a que un entero cuyo valor es cero es un subconjunto de todos los enteros, que es el patrón definido por la instrucción `case int val`.

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Example
{
   public static void Main()
   {
      var values = new List<object>();
      for (int ctr = 0; ctr <= 7; ctr++) {
         if (ctr == 2)
            values.Add(DiceLibrary.Roll2());
         else if (ctr == 4)
            values.Add(DiceLibrary.Pass());
         else
            values.Add(DiceLibrary.Roll());
      }

      Console.WriteLine($"The sum of { values.Count } die is { DiceLibrary.DiceSum(values) }");
   }
}

public static class DiceLibrary
{
   // Random number generator to simulate dice rolls.
   static Random rnd = new Random();

   // Roll a single die.
   public static int Roll()
   {
      return rnd.Next(1, 7);
   }

   // Roll two dice.
   public static List<object> Roll2()
   {
      var rolls = new List<object>();
      rolls.Add(Roll());
      rolls.Add(Roll());
      return rolls;
   }

   // Calculate the sum of n dice rolls.
   public static int DiceSum(IEnumerable<object> values)
   {
      var sum = 0;
      foreach (var item in values)
      {
            switch (item)
            {
               // A single zero value.
               case 0:
                  break;
```

```
                // A single value.
                case int val:
                   sum += val;
                   break;
                // A non-empty collection.
                case IEnumerable<object> subList when subList.Any():
                   sum += DiceSum(subList);
                   break;
                // An empty collection.
                case IEnumerable<object> subList:
                   break;
                //  A null reference.
                case null:
                   break;
                // A value that is neither an integer nor a collection.
                default:
                   throw new InvalidOperationException("unknown item type");
             }
       }
       return sum;
    }

    public static object Pass()
    {
       if (rnd.Next(0, 2) == 0)
          return null;
       else
          return new List<object>();
    }
 }
```

Puede corregir este problema y eliminar la advertencia del compilador de alguna de estas dos formas:

- Si cambia el orden de las secciones switch.
- Si usa una cláusula when en la etiqueta `case`.

Etiqueta case `default`

La etiqueta case `default` especifica la sección switch que se va a ejecutar si la expresión de coincidencia no coincide con ninguna otra etiqueta `case`. Si no hay ninguna etiqueta case `default` y la expresión de coincidencia no coincide con ninguna otra etiqueta `case`, el flujo del programa pasa a la instrucción `switch`.

La etiqueta case `default` puede aparecer en cualquier orden en la instrucción `switch`. Independientemente de su orden en el código fuente, siempre se evalúa en último lugar, después de que se hayan evaluado las demás etiquetas `case`.

Coincidencia de patrones con la instrucción `switch`

Cada instrucción `case` define un patrón que, si coincide con la expresión de coincidencia, provoca la ejecución de su sección switch contenedora. Todas las versiones de C# admiten el patrón de constante. Los demás patrones se admiten a partir de C# 7.0.

Patrón de constante

El patrón de constante comprueba si la expresión de coincidencia es igual a una constante especificada. Su sintaxis es:

```
    case constant:
```

donde *constant* es el valor que se va a comprobar. *constant* puede ser cualquiera de las expresiones de constante siguientes:

- Un literal booleano, ya sea `true` o `false`.
- Cualquier constante entera, como `int`, `long` o `byte`.
- El nombre de una variable `const` declarada.
- Una constante de enumeración.
- Un literal de carácter.
- Un literal de cadena.

La expresión de constante se evalúa de la siguiente forma:

- Si *expr* y *constant* son tipos enteros, el operador de igualdad de C# determina si la expresión devuelve `true` (es decir, si `expr == constant`).
- De lo contrario, el valor de la expresión se determina mediante una llamada al método estático Object.Equals(expr, constant).

En el ejemplo siguiente se usa el patrón de constante para determinar si una fecha determinada es un fin de semana, el primer día, el último día o la mitad de la semana laboral. Evalúa la propiedad DateTime.DayOfWeek del día actual con los miembros de la enumeración DayOfWeek.

```
using System;

class Program
{
    static void Main()
    {
        switch (DateTime.Now.DayOfWeek)
        {
           case DayOfWeek.Sunday:
           case DayOfWeek.Saturday:
              Console.WriteLine("The weekend");
              break;
           case DayOfWeek.Monday:
              Console.WriteLine("The first day of the work week.");
              break;
           case DayOfWeek.Friday:
              Console.WriteLine("The last day of the work week.");
              break;
           default:
              Console.WriteLine("The middle of the work week.");
              break;
        }
    }
}
// The example displays output like the following:
//       The middle of the work week.
```

En el ejemplo siguiente se usa el patrón de constante para controlar la entrada del usuario en una aplicación de consola que simula una cafetera automática.

```
using System;

class Example
{
   static void Main()
   {
       Console.WriteLine("Coffee sizes: 1=small 2=medium 3=large");
       Console.Write("Please enter your selection: ");
       string str = Console.ReadLine();
       int cost = 0;

       // Because of the goto statements in cases 2 and 3, the base cost of 25
       // cents is added to the additional cost for the medium and large sizes.
       switch (str)
       {
          case "1":
          case "small":
              cost += 25;
              break;
          case "2":
          case "medium":
              cost += 25;
              goto case "1";
          case "3":
          case "large":
              cost += 50;
              goto case "1";
          default:
              Console.WriteLine("Invalid selection. Please select 1, 2, or 3.");
              break;
      }
      if (cost != 0)
      {
          Console.WriteLine("Please insert {0} cents.", cost);
      }
      Console.WriteLine("Thank you for your business.");
   }
}
// The example displays output like the following:
//         Coffee sizes: 1=small 2=medium 3=large
//         Please enter your selection: 2
//         Please insert 50 cents.
//         Thank you for your business.
```

Patrón de tipo

El patrón de tipo habilita la conversión y la evaluación de tipo concisas. Cuando se usa con la instrucción `switch` para realizar la coincidencia de patrones, comprueba si una expresión se puede convertir en un tipo especificado y, en caso afirmativo, la convierte en una variable de ese tipo. Su sintaxis es:

```
   case type varname
```

donde *type* es el nombre del tipo al que se va a convertir el resultado de *expr* y *varname* es el objeto al que se va a convertir el resultado de *expr* si hay coincidencia. El tipo de tiempo de compilación de *expr* puede ser un parámetro de tipo genérico a partir de C# 7.1.

La expresión `case` es `true` si se cumple alguna de las siguientes condiciones:

- *expr* es una instancia del mismo tipo que *type*.
- *expr* es una instancia de un tipo que deriva de *type*. En otras palabras, el resultado de *expr* puede convertirse en una instancia de *type*.

- *expr* tiene un tipo en tiempo de compilación que es una clase base de *type* y *expr* tiene un tipo en tiempo de ejecución que es *type* o se deriva de *type*. El *tipo en tiempo de compilación* de una variable es el tipo de la variable tal como se define en su declaración de tipos. El *tipo en tiempo de ejecución* de una variable es el tipo de la instancia que se asigna a esa variable.
- *type* es una instancia de un tipo que implementa la interfaz *type*.

Si la expresión case es true, *varname* se asigna definitivamente y tiene ámbito local únicamente dentro de la sección switch.

Tenga en cuenta que `null` no coincide con un tipo. Para que `null` coincida, use la siguiente etiqueta `case`:

```
case null:
```

En el ejemplo siguiente se usa el patrón de tipo para proporcionar información sobre los distintos tipos de colección.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        switch (coll)
        {
            case Array arr:
               Console.WriteLine($"An array with {arr.Length} elements.");
               break;
            case IEnumerable<int> ieInt:
               Console.WriteLine($"Average: {ieInt.Average(s => s)}");
               break;
            case IList list:
               Console.WriteLine($"{list.Count} items");
               break;
            case IEnumerable ie:
               string result = "";
               foreach (var e in ie)
                  result += $"{e} ";
               Console.WriteLine(result);
               break;
            case null:
               // Do nothing for a null.
               break;
            default:
               Console.WriteLine($"A instance of type {coll.GetType().Name}");
               break;
        }
    }
}
// The example displays the following output:
//     An array with 5 elements.
//     4 items
```

En lugar de `object`, podría crear un método genérico, con el tipo de la colección como el parámetro de tipo, tal como se muestra en el código siguiente:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation<T>(T coll)
    {
        switch (coll)
        {
            case Array arr:
               Console.WriteLine($"An array with {arr.Length} elements.");
               break;
            case IEnumerable<int> ieInt:
               Console.WriteLine($"Average: {ieInt.Average(s => s)}");
               break;
            case IList list:
               Console.WriteLine($"{list.Count} items");
               break;
            case IEnumerable ie:
               string result = "";
               foreach (var e in ie)
                  result += $"{e} ";
               Console.WriteLine(result);
               break;
            case object o:
               Console.WriteLine($"A instance of type {o.GetType().Name}");
               break;
            default:
                Console.WriteLine("Null passed to this method.");
                break;
        }
    }
}
// The example displays the following output:
//     An array with 5 elements.
//     4 items
//     Null passed to this method.
```

La versión genérica es distinta del primer ejemplo de dos maneras. En primer lugar, no puede usar el caso `null`. No puede usar ningún caso constante porque el compilador no puede convertir ningún tipo arbitrario `T` a ningún tipo distinto de `object`. Lo que habría sido el caso `default` ahora se prueba para un `object` no nulo. Esto significa que el caso `default` solo se prueba para `null`.

Sin coincidencia de patrones, este código podría escribirse del modo siguiente. El uso de la coincidencia de patrones de tipo genera código más compacto y legible al eliminar la necesidad de comprobar si el resultado de una conversión es `null` o de realizar conversiones repetidas.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        if (coll is Array)
        {
           Array arr = (Array) coll;
           Console.WriteLine($"An array with {arr.Length} elements.");
        }
        else if (coll is IEnumerable<int>)
        {
            IEnumerable<int> ieInt = (IEnumerable<int>) coll;
            Console.WriteLine($"Average: {ieInt.Average(s => s)}");
        }
        else if (coll is IList)
        {
            IList list = (IList) coll;
            Console.WriteLine($"{list.Count} items");
        }
        else if (coll is IEnumerable)
        {
            IEnumerable ie = (IEnumerable) coll;
            string result = "";
            foreach (var e in ie)
               result += $"{e} ";
            Console.WriteLine(result);
        }
        else if (coll == null)
        {
            // Do nothing.
        }
        else
        {
            Console.WriteLine($"An instance of type {coll.GetType().Name}");
        }
    }
}
// The example displays the following output:
//     An array with 5 elements.
//     4 items
```

Instrucción `case` y cláusula `when`

A partir de C# 7.0, dado que las instrucciones case no necesitan ser mutuamente excluyentes, puede agregar una cláusula `when` para especificar una condición adicional que deba cumplirse para que la instrucción case se evalúe como true. La cláusula `when` puede ser cualquier expresión que devuelva un

valor booleano.

En el ejemplo siguiente se define una clase base `Shape`, una clase `Rectangle` que deriva de `Shape` y una clase `Square` que deriva de `Rectangle`. Usa la cláusula `when` para asegurarse de que `ShowShapeInfo` trate a un objeto `Rectangle` al que se han asignado las mismas longitudes y anchos como si fuera `Square` aunque de él no se hayan creado instancias como de un objeto `Square`. El método no intenta mostrar información sobre un objeto que es `null` ni sobre una forma cuya área es cero.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }
   public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public double Length { get; set; }
   public double Width { get; set; }

   public override double Area
   {
      get { return Math.Round(Length * Width,2); }
   }

   public override double Circumference
   {
      get { return (Length + Width) * 2; }
   }
}

public class Square : Rectangle
{
   public Square(double side) : base(side, side)
   {
      Side = side;
   }

   public double Side { get; set; }
}

public class Circle : Shape
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public override double Circumference
   {
      get { return 2 * Math.PI * Radius; }
   }

   public override double Area
   {
      get { return Math.PI * Math.Pow(Radius, 2); }
   }
```

```
}

public class Example
{
   public static void Main()
   {
      Shape sh = null;
      Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                         sh, new Square(0), new Rectangle(8, 8),
                         new Circle(3) };
      foreach (var shape in shapes)
         ShowShapeInfo(shape);
   }

   private static void ShowShapeInfo(Shape sh)
   {
      switch (sh)
      {
         // Note that this code never evaluates to true.
         case Shape shape when shape == null:
            Console.WriteLine($"An uninitialized shape (shape == null)");
            break;
         case null:
            Console.WriteLine($"An uninitialized shape");
            break;
         case Shape shape when sh.Area == 0:
            Console.WriteLine($"The shape: {sh.GetType().Name} with no dimensions");
            break;
         case Square sq when sh.Area > 0:
            Console.WriteLine("Information about square:");
            Console.WriteLine($"   Length of a side: {sq.Side}");
            Console.WriteLine($"   Area: {sq.Area}");
            break;
         case Rectangle r when r.Length == r.Width && r.Area > 0:
            Console.WriteLine("Information about square rectangle:");
            Console.WriteLine($"   Length of a side: {r.Length}");
            Console.WriteLine($"   Area: {r.Area}");
            break;
         case Rectangle r when sh.Area > 0:
            Console.WriteLine("Information about rectangle:");
            Console.WriteLine($"   Dimensions: {r.Length} x {r.Width}");
            Console.WriteLine($"   Area: {r.Area}");
            break;
         case Shape shape when sh != null:
            Console.WriteLine($"A {sh.GetType().Name} shape");
            break;
         default:
            Console.WriteLine($"The {nameof(sh)} variable does not represent a Shape.");
            break;
      }
   }
}
// The example displays the following output:
//       Information about square:
//          Length of a side: 10
//          Area: 100
//       Information about rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//       An uninitialized shape
//       The shape: Square with no dimensions
//       Information about square rectangle:
//          Length of a side: 8
//          Area: 64
//       A Circle shape
```

Tenga en cuenta que la cláusula `when` del ejemplo que intenta comprobar si un objeto `Shape` es `null`

no se ejecuta. El patrón de tipo correcto para comprobar `null` es `case null:`.

Especificación del lenguaje C#

Para obtener más información, vea el apartado Instrucción switch en Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- if-else
- Coincidencia de patrones

do (Referencia de C#)

29/06/2020 • 2 minutes to read • Edit Online

La instrucción `do` ejecuta una instrucción o un bloque de instrucciones mientras que una expresión booleana especificada se evalúa como `true`. Como esa expresión se evalúa después de cada ejecución del bucle, un bucle `do-while` se ejecuta una o varias veces. Esto es diferente del bucle while, que se ejecuta cero o varias veces.

En cualquier punto del bloque de instrucciones `do`, se puede salir del bucle mediante la instrucción break.

Puede ir directamente a la evaluación de la expresión `while` mediante la instrucción continue. Si la expresión se evalúa como `true`, la ejecución continúa en la primera instrucción del bucle. En caso contrario, la ejecución continúa en la primera instrucción después del bucle.

También se puede salir de un bucle `do-while` mediante las instrucciones goto, return o throw.

Ejemplo

En el ejemplo siguiente se muestra el uso de la instrucción `do`. Haga clic en **Ejecutar** para ejecutar el código de ejemplo. Después, puede modificar el código y volver a ejecutarlo.

```
int n = 0;
do
{
    Console.WriteLine(n);
    n++;
} while (n < 5);
```

Especificación del lenguaje C#

Para más información, vea la sección sobre la instrucción do de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- while (Instrucción)

for (Referencia de C#)

29/06/2020 • 5 minutes to read • Edit Online

La instrucción `for` ejecuta una instrucción o un bloque de instrucciones mientras una expresión booleana especificada se evalúa como `true`.

En cualquier punto del bloque de instrucciones `for`, se puede salir del bucle mediante la instrucción break, o bien se puede ir a la siguiente iteración del bucle mediante la instrucción continue. También se puede salir de un bucle `for` mediante las instrucciones goto, return o throw.

Estructura de la instrucción `for`

La instrucción `for` define las secciones *inicializador*, *condición* e *iterador*:

```
for (initializer; condition; iterator)
    body
```

Las tres secciones son opcionales. El cuerpo del bucle es una instrucción o un bloque de instrucciones.

En el siguiente ejemplo se muestra la instrucción `for` con todas las secciones definidas:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

La sección *inicializador*

Las instrucciones de la sección *inicializador* se ejecutan solo una vez, antes de entrar en el bucle. La sección *inicializador* es cualquiera de las siguientes:

- La declaración e inicialización de una variable de bucle local, a la que no se puede tener acceso desde fuera del bucle.
- Ninguna, una o varias expresiones de instrucción de la siguiente lista, separadas por comas:
  - instrucción assignment
  - invocación de un método
  - expresión de incremento de prefijo o sufijo, como `++i` o `i++`
  - expresión de decremento de prefijo o sufijo, como `--i` o `i--`
  - creación de un objeto mediante el operador new
  - expresión await

La sección *inicializador* del ejemplo anterior declara e inicializa la variable de bucle local `i`:

```
int i = 0
```

La sección *condición*

La sección *condición*, si está presente, debe ser una expresión booleana. Dicha expresión se evalúa antes de cada iteración del bucle. Si la sección *condición* no está presente o la expresión booleana se evalúa como `true`, se ejecutará la siguiente iteración del bucle; en caso contrario, se sale del bucle.

La sección *condición* del ejemplo anterior determina si el bucle finaliza en función del valor de la variable de bucle local:

```
i < 5
```

La sección *iterador*

La sección *iterador* define lo que sucede después de cada iteración del cuerpo del bucle. La sección *iterador* contiene ninguna o más de las siguientes expresiones de instrucción, separadas por comas:

- instrucción assignment
- invocación de un método
- expresión de incremento de prefijo o sufijo, como `++i` o `i++`
- expresión de decremento de prefijo o sufijo, como `--i` o `i--`
- creación de un objeto mediante el operador new
- expresión await

La sección *iterador* del ejemplo anterior incrementa la variable de bucle local:

```
i++
```

Ejemplos

En el ejemplo siguiente se muestran varios usos menos comunes de las secciones de la instrucción `for`: asignar un valor a una variable de bucle externa en la sección *inicializador*, invocar un método en las secciones *inicializador* e *iterador*, y cambiar los valores de dos variables en la sección *iterador*. Haga clic en **Ejecutar** para ejecutar el código de ejemplo. Después, puede modificar el código y volver a ejecutarlo.

```
int i;
int j = 10;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--, Console.WriteLine($"Step: i=
{i}, j={j}"))
{
    // Body of the loop.
}
```

En el ejemplo siguiente se define el bucle `for` infinito:

```
for ( ; ; )
{
    // Body of the loop.
}
```

Especificación del lenguaje C#

Para más información, vea la sección La declaración for de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- foreach, in

foreach, in (Referencia de C#)

29/06/2020 • 7 minutes to read • Edit Online

La instrucción `foreach` ejecuta una instrucción o un bloque de instrucciones para cada elemento en una instancia del tipo que implementa la interfaz System.Collections.IEnumerable o System.Collections.Generic.IEnumerable<T>. La instrucción `foreach` no se limita a esos tipos y puede aplicarse a una instancia de cualquier tipo que cumpla las siguientes condiciones:

- tiene el método público sin parámetros `GetEnumerator` cuyo tipo de valor devuelto es clase, estructura o tipo de interfaz,
- el tipo de valor devuelto del método `GetEnumerator` tiene la propiedad pública `Current` y el método público sin parámetros `MoveNext` cuyo tipo de valor devuelto es Boolean.

En la mayoría de casos, `foreach` itera una expresión `IEnumerable<T>` en la que cada elemento es de tipo `T`. Pero los elementos pueden ser de cualquier tipo que tenga una conversión implícita o explícita del tipo de la propiedad `Current`. Si la propiedad `Current` devuelve `SomeType`, el tipo de los elementos puede ser uno de los siguientes:

- Cualquiera de las clases base de `SomeType`.
- Cualquiera de las interfaces implementadas por `SomeType`.

Además, si `SomeType` es una `class` o una `interface` y no tiene el estado `sealed`, el tipo de elemento puede ser uno de los siguientes:

- Cualquier tipo derivado de `SomeType`.
- Cualquier interfaz arbitraria. Se permite cualquier interfaz porque se puede usar una clase derivada de `SomeType` o que implemente este elemento para implementar cualquier interfaz.

Puede declarar la variable de iteración con cualquier tipo que coincida con las reglas anteriores. Si la conversión de `SomeType` al tipo de la variable de iteración requiere una conversión explícita, dicha operación puede producir una excepción InvalidCastException al producirse un error de conversión.

A partir de C# 7.3, si la propiedad `Current` del enumerador devuelve un valor devuelto de referencia (`ref T` donde `T` es el tipo del elemento de colección), se puede declarar la variable de iteración con el modificador `ref` o `ref readonly`.

A partir de C# 8.0, el operador `await` se puede aplicar a la instrucción `foreach` cuando el tipo de colección implemente la interfaz IAsyncEnumerable<T>. Cada iteración del bucle se puede suspender mientras el siguiente elemento se recupera de forma asincrónica. Los elementos de secuencia se procesan de forma predeterminada en el contexto capturado. Si quiere deshabilitar la captura del contexto, use el método de extensión TaskAsyncEnumerableExtensions.ConfigureAwait. Para obtener más información sobre los contextos de sincronización y la captura del contexto actual, vea el artículo sobre el consumo del patrón asincrónico basado en tareas.

En cualquier punto del bloque de instrucciones `foreach`, se puede salir del bucle mediante la instrucción break, o bien se puede ir a la siguiente iteración del bucle mediante la instrucción continue. También se puede salir de un bucle `foreach` mediante las instrucciones goto, return o throw.

Si la instrucción `foreach` se aplica a `null`, se produce NullReferenceException. Si la colección de origen de la instrucción `foreach` está vacía, el cuerpo del bucle `foreach` no se ejecuta y se omite.

Ejemplos

> **NOTE**
>
> Los ejemplos de C# de este artículo se ejecutan en el ejecutor de código en línea y área de juegos de Try.NET. Haga clic en el botón **Ejecutar** para ejecutar un ejemplo en una ventana interactiva. Una vez que se ejecuta el código, puede modificar y ejecutar el código modificado si vuelve a hacer clic en **Ejecutar**. El código modificado se ejecuta en la ventana interactiva o, si se produce un error en la compilación, en la ventana interactiva se muestran todos los mensajes de error del compilador de C#.

El siguiente ejemplo muestra el uso de la instrucción `foreach` con una instancia del tipo List<T> que implementa la interfaz IEnumerable<T>:

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
int count = 0;
foreach (int element in fibNumbers)
{
    count++;
    Console.WriteLine($"Element #{count}: {element}");
}
Console.WriteLine($"Number of elements: {count}");
```

El siguiente ejemplo utiliza la instrucción `foreach` con una instancia del tipo System.Span<T>, que no implementa ninguna interfaz:

```
public class IterateSpanExample
{
    public static void Main()
    {
        Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
        foreach (int number in numbers)
        {
            Console.Write($"{number} ");
        }
        Console.WriteLine();
    }
}
```

En el ejemplo siguiente se usa una variable de iteración `ref` para establecer el valor de cada elemento de una matriz stackalloc. La versión `ref readonly` recorre en iteración la colección para imprimir todos los valores. En la declaración de `readonly`, se usa una declaración de variable local implícita. Las declaraciones de variables implícitas se pueden usar con las declaraciones `ref` o `ref readonly`, al igual que las declaraciones de variables con tipo explícito.

```
public class ForeachRefExample
{
    public static void Main()
    {
        Span<int> storage = stackalloc int[10];
        int num = 0;
        foreach (ref int item in storage)
        {
            item = num++;
        }

        foreach (ref readonly var item in storage)
        {
            Console.Write($"{item} ");
        }
        // Output:
        // 0 1 2 3 4 5 6 7 8 9
    }
}
```

En el siguiente ejemplo se usa `await foreach` para iterar por una colección que genera cada elemento de forma asincrónica:

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

Especificación del lenguaje C#

Para más información, vea la sección sobre la instrucción foreach de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Utilizar foreach con matrices
- for (Instrucción)

while (Referencia de C#)

29/06/2020 • 2 minutes to read • Edit Online

La instrucción `while` ejecuta una instrucción o un bloque de instrucciones mientras que una expresión booleana especificada se evalúa como `true`. Como esa expresión se evalúa antes de cada ejecución del bucle, un bucle `while` se ejecuta cero o varias veces. Esto es diferente del bucle do que se ejecuta una o varias veces.

En cualquier punto del bloque de instrucciones `while`, se puede salir del bucle mediante la instrucción break.

Puede ir directamente a la evaluación de la expresión `while` mediante la instrucción continue. Si la expresión se evalúa como `true`, la ejecución continúa en la primera instrucción del bucle. En caso contrario, la ejecución continúa en la primera instrucción después del bucle.

También se puede salir de un bucle `while` mediante las instrucciones goto, return o throw.

Ejemplo

En el ejemplo siguiente se muestra el uso de la instrucción `while`. Haga clic en **Ejecutar** para ejecutar el código de ejemplo. Después, puede modificar el código y volver a ejecutarlo.

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Especificación del lenguaje C#

Para más información, vea la sección La instrucción while de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- do (instrucción)

break (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

La instrucción `break` finaliza la ejecución del bucle contenedor más próximo o de la instrucción switch en la que aparezca. El control se pasa a la instrucción que hay a continuación de la instrucción finalizada, si existe.

Ejemplo

En este ejemplo, la instrucción condicional contiene un contador que se supone que cuenta de 1 a 100. Pero la instrucción `break` finaliza el bucle después de cuatro recuentos.

```
class BreakTest
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
 Output:
    1
    2
    3
    4
*/
```

Ejemplo

En este ejemplo se muestra el uso de `break` en una instrucción switch.

```
class Switch
{
    static void Main()
    {
        Console.Write("Enter your selection (1, 2, or 3): ");
        string s = Console.ReadLine();
        int n = Int32.Parse(s);

        switch (n)
        {
            case 1:
                Console.WriteLine("Current value is 1");
                break;
            case 2:
                Console.WriteLine("Current value is 2");
                break;
            case 3:
                Console.WriteLine("Current value is 3");
                break;
            default:
                Console.WriteLine("Sorry, invalid selection.");
                break;
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 1

Sample Output:
Enter your selection (1, 2, or 3): 1
Current value is 1
*/
```

Si se escribió `4`, la salida será:

```
Enter your selection (1, 2, or 3): 4
Sorry, invalid selection.
```

Ejemplo

En este ejemplo, se usa la instrucción `break` para salir de un bucle anidado interno y devolver el control al bucle externo. El control *solo* se devuelve un nivel hacia arriba en los bucles anidados.

```
class BreakInNestedLoops
{
    static void Main(string[] args)
    {

        int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };

        // Outer loop.
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine($"num = {numbers[i]}");

            // Inner loop.
            for (int j = 0; j < letters.Length; j++)
            {
                if (j == i)
                {
                    // Return control to outer loop.
                    break;
                }
                Console.Write($" {letters[j]} ");
            }
            Console.WriteLine();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
    num = 0

    num = 1
     a
    num = 2
     a  b
    num = 3
     a  b  c
    num = 4
     a  b  c  d
    num = 5
     a  b  c  d  e
    num = 6
     a  b  c  d  e  f
    num = 7
     a  b  c  d  e  f  g
    num = 8
     a  b  c  d  e  f  g  h
    num = 9
     a  b  c  d  e  f  g  h  i
 */
```

Ejemplo

En este ejemplo, la instrucción `break` solo se usa para salir de la rama actual durante cada iteración del bucle. El propio bucle no se ve afectado por las instancias de `break` que pertenecen a la instrucción anidada switch.

```
class BreakFromSwitchInsideLoop
{
    static void Main(string[] args)
    {
        // loop 1 to 3
        for (int i = 1; i <= 3; i++)
        {
            switch(i)
            {
                case 1:
                    Console.WriteLine("Current value is 1");
                    break;
                case 2:
                    Console.WriteLine("Current value is 2");
                    break;
                case 3:
                    Console.WriteLine("Current value is 3");
                    break;
                default:
                    Console.WriteLine("This shouldn't happen.");
                    break;
            }
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
    Current value is 1
    Current value is 2
    Current value is 3
 */
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- switch

continue (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La instrucción `continue` transfiere el control a la siguiente iteración de la instrucción envolvente while, do, for o foreach en la que aparece.

Ejemplo

En este ejemplo se inicializa un contador para contar del 1 al 10. Si se usa la instrucción `continue` junto con la expresión `(i < 9)`, se omiten las instrucciones comprendidas entre `continue` y el final del cuerpo de `for`.

```
class ContinueTest
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
9
10
*/
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- break (Instrucción)

goto (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La instrucción `goto` transfiere el control del programa directamente a una instrucción con etiqueta.

Un uso común de `goto` consiste en transferir el control a una etiqueta de switch case específica o a la etiqueta predeterminada en una instrucción `switch`.

La instrucción `goto` también es útil para salir de bucles demasiado anidados.

Ejemplo

En el ejemplo siguiente se muestra cómo usar `goto` en una instrucción switch.

```
class SwitchTest
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
        Console.Write("Please enter your selection: ");
        string s = Console.ReadLine();
        int n = int.Parse(s);
        int cost = 0;
        switch (n)
        {
            case 1:
                cost += 25;
                break;
            case 2:
                cost += 25;
                goto case 1;
            case 3:
                cost += 50;
                goto case 1;
            default:
                Console.WriteLine("Invalid selection.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine($"Please insert {cost} cents.");
        }
        Console.WriteLine("Thank you for your business.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input:  2

Sample Output:
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.
*/
```

Ejemplo

En el ejemplo siguiente se muestra cómo usar `goto` para salir de bucles anidados.

```
public class GotoTest1
{
    static void Main()
    {
        int x = 200, y = 4;
        int count = 0;
        string[,] array = new string[x, y];

        // Initialize the array.
        for (int i = 0; i < x; i++)

            for (int j = 0; j < y; j++)
                array[i, j] = (++count).ToString();

        // Read input.
        Console.Write("Enter the number to search for: ");

        // Input a string.
        string myNumber = Console.ReadLine();

        // Search.
        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < y; j++)
            {
                if (array[i, j].Equals(myNumber))
                {
                    goto Found;
                }
            }
        }

        Console.WriteLine($"The number {myNumber} was not found.");
        goto Finish;

    Found:
        Console.WriteLine($"The number {myNumber} is found.");

    Finish:
        Console.WriteLine("End of search.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 44

Sample Output
Enter the number to search for: 44
The number 44 is found.
End of search.
*/
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- goto (Instrucción) (C++)

return (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La instrucción `return` termina la ejecución del método en el que aparece y devuelve el control al método de llamada. También puede devolver un valor opcional. Si el método es del tipo `void`, la instrucción `return` se puede omitir.

Si la instrucción return está incluida en un bloque `try`, el bloque `finally`, si existe, se ejecutará antes de que el control se devuelva al método de llamada.

Ejemplo

En el siguiente ejemplo, el método `CalculateArea()` devuelve la variable local `area` como un valor de tipo `double`.

```
class ReturnTest
{
    static double CalculateArea(int r)
    {
        double area = r * r * Math.PI;
        return area;
    }

    static void Main()
    {
        int radius = 5;
        double result = CalculateArea(radius);
        Console.WriteLine("The area is {0:0.00}", result);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: The area is 78.54
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- return (instrucción)

throw (Referencia de C#)

18/03/2020 • 5 minutes to read • Edit Online

Indica la aparición de una excepción durante la ejecución del programa.

Comentarios

La sintaxis de `throw` es la siguiente:

```
throw [e];
```

donde `e` es una instancia de una clase derivada de System.Exception. En el ejemplo siguiente se usa la instrucción `throw` para producir una excepción IndexOutOfRangeException si el argumento pasado a un método denominado `GetNumber` no se corresponde con un índice válido de una matriz interna.

```
using System;

public class NumberGenerator
{
   int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

   public int GetNumber(int index)
   {
      if (index < 0 || index >= numbers.Length) {
         throw new IndexOutOfRangeException();
      }
      return numbers[index];
   }
}
```

Después, los autores de llamadas a método usan un bloque `try-catch` o `try-catch-finally` para controlar la excepción generada. En el ejemplo siguiente se controla la excepción producida por el método `GetNumber`.

```
using System;

public class Example
{
   public static void Main()
   {
      var gen = new NumberGenerator();
      int index = 10;
      try {
          int value = gen.GetNumber(index);
          Console.WriteLine($"Retrieved {value}");
      }
      catch (IndexOutOfRangeException e)
      {
         Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
      }
   }
}
// The example displays the following output:
//        IndexOutOfRangeException: 10 is outside the bounds of the array
```

Volver a iniciar una excepción

`throw` también se puede usar en un bloque `catch` para volver a iniciar una excepción controlada en un bloque `catch`. En este caso, `throw` no toma un operando de excepción. Resulta más útil cuando un método pasa un argumento de un autor de llamada a otro método de biblioteca y el método de biblioteca produce una excepción que se debe pasar al autor de llamada. Por ejemplo, en el ejemplo siguiente se vuelve a iniciar una excepción NullReferenceException que se produce al intentar recuperar el primer carácter de una cadena sin inicializar.

```
using System;

public class Sentence
{
   public Sentence(string s)
   {
      Value = s;
   }

   public string Value { get; set; }

   public char GetFirstCharacter()
   {
      try {
         return Value[0];
        }
      catch (NullReferenceException e) {
         throw;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var s = new Sentence(null);
      Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
   }
}
// The example displays the following output:
//    Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
object.
//       at Sentence.GetFirstCharacter()
//       at Example.Main()
```

IMPORTANT

También puede usar la sintaxis `throw e` en un bloque `catch` para crear instancias de una nueva excepción que se pase al autor de llamada. En este caso, no se conserva el seguimiento de la pila de la excepción original, que está disponible en la propiedad StackTrace.

La expresión `throw`

A partir de C# 7.0, se puede usar `throw` como una expresión y como una instrucción. Esto permite iniciar una excepción en contextos que antes no se admitían. Se incluyen los siguientes:

- El operador condicional. En el ejemplo siguiente se usa una expresión `throw` para iniciar una excepción ArgumentException si se pasa a un método una matriz de cadena vacía. Antes de C# 7.0, esta lógica tenía que aparecer en una instrucción `if` / `else`.

```
private static void DisplayFirstNumber(string[] args)
{
   string arg = args.Length >= 1 ? args[0] :
                              throw new ArgumentException("You must supply an argument");
   if (Int64.TryParse(arg, out var number))
      Console.WriteLine($"You entered {number:F0}");
   else
      Console.WriteLine($"{arg} is not a number.");
}
```

- El operador de uso combinado de NULL. En el ejemplo siguiente, se usa una expresión `throw` con un operador de uso combinado de NULL para producir una excepción si la cadena asignada a una propiedad `Name` es `null`.

```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "Name cannot be null");
}
```

- Un método o lambda con forma de expresión. En el ejemplo siguiente se muestra un método con forma de expresión que produce una excepción InvalidCastException porque no se admite una conversión a un valor DateTime.

```
DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is not supported.");
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- try-catch
- Palabras clave de C#
- Cómo: Iniciar excepciones explícitamente

try-catch (Referencia de C#)

18/03/2020 • 15 minutes to read • Edit Online

La instrucción try-catch consta de un bloque `try` seguido de una o más cláusulas `catch` que especifican controladores para diferentes excepciones.

Cuando se produce una excepción, Common Language Runtime (CLR) busca la instrucción `catch` que controla esta excepción. Si el método que se ejecuta actualmente no contiene un bloque `catch`, CLR busca el método que llamó el método actual, y así sucesivamente hasta la pila de llamadas. Si no existe ningún bloque `catch`, CLR muestra al usuario un mensaje de excepción no controlada y detiene la ejecución del programa.

El bloque `try` contiene el código protegido que puede producir la excepción. El bloque se ejecuta hasta que se produce una excepción o hasta que se completa correctamente. Por ejemplo, el intento siguiente de convertir un objeto `null` produce la excepción NullReferenceException:

```
object o2 = null;
try
{
    int i2 = (int)o2;   // Error
}
```

Aunque la cláusula `catch` puede utilizarse sin argumentos para detectar cualquier tipo de excepción, no se recomienda este uso. En general, solo debe convertir las excepciones que sabe cómo recuperar. Por lo tanto, debe especificar siempre un argumento de objeto derivado de System.Exception Por ejemplo:

```
catch (InvalidCastException e)
{
}
```

Es posible utilizar más de una cláusula `catch` específica en la misma instrucción try-catch. En este caso, el orden de las cláusulas `catch` es importante, puesto que las cláusulas `catch` se examinan por orden. Detectar las excepciones más específicas antes que las menos específicas. El compilador genera un error si ordena los bloques de detección para que un bloque posterior nunca pueda alcanzarse.

La utilización de los argumentos `catch` es una manera de filtrar las excepciones que desea controlar. También se puede usar una expresión de filtro que examine aún más la excepción para decidir si controlarla. Si la expresión de filtro devuelve false, prosigue la búsqueda de un controlador.

```
catch (ArgumentException e) when (e.ParamName == "…")
{
}
```

Los filtros de excepción son preferibles para detectar y volver a producir (se explica a continuación) porque los filtros dejan la pila intacta. Si un controlador posterior vuelca la pila, puede ver la procedencia original de la excepción, más que solo la ubicación en la que se volvió a producir. Un uso común de las expresiones de filtro de excepciones es el registro. Puede crear una función de filtro que siempre devuelva false y que también resulte en un registro, o bien puede registrar excepciones a medida que se produzcan sin tener que controlarlas y volver a generarlas.

Se puede usar una instrucción throw en un bloque `catch` para volver a iniciar la excepción detectada por la instrucción `catch`. En el ejemplo siguiente se extrae información de origen de una excepción IOException y, a continuación, se produce la excepción al método principal.

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

Puede capturar una excepción y producir una excepción diferente. Al hacerlo, especifique la excepción que detectó como excepción interna, tal como se muestra en el ejemplo siguiente.

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

También se puede volver a producir una excepción sin una condición específica es true, tal y como se muestra en el ejemplo siguiente.

```
catch (InvalidCastException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}
```

NOTE

También es posible usar un filtro de excepción para obtener un resultado similar de una forma generalmente más limpia (además de no modificar la pila, tal y como se explicó anteriormente en este documento). El ejemplo siguiente tiene un comportamiento similar para los autores de llamada que el ejemplo anterior. La función inicia la excepción `InvalidCastException` de vuelta al autor de la llamada cuando `e.Data` es `null`.

```
catch (InvalidCastException e) when (e.Data != null)
{
    // Take some action.
}
```

Desde dentro de un bloque `try`, solo deben inicializarse las variables que se declaran en el mismo. De lo contrario, puede ocurrir una excepción antes de que se complete la ejecución del bloque. Por ejemplo, en el siguiente ejemplo de código, la variable `n` se inicializa dentro del bloque `try`. Un intento de utilizar esta

variable fuera del bloque `try` en la instrucción `Write(n)` generará un error del compilador.

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.Write(n);
}
```

Para obtener más información sobre la captura,vea try-catch-finally (try-catch-finally [Referencia de C#]).

Excepciones en métodos asincrónicos

Un método asincrónico está marcado por un modificador async y normalmente contiene una o más instrucciones o expresiones await. Una expresión await aplica el operador await a Task o Task<TResult>.

Cuando el control alcanza un `await` en el método asincrónico, el progreso del método se suspende hasta que la tarea esperada se completa. Cuando se completa la tarea, la ejecución puede reanudarse en el método. Para más información, vea Programación asincrónica con Async y Await y Controlar el flujo en los programas asincrónicos.

La tarea completada a la que se aplica `await` puede encontrarse en un estado de error debido a una excepción no controlada en el método que devuelve la tarea. La espera de la tarea produce una excepción. Una tarea también puede terminar en un estado cancelado si se cancela el proceso asincrónico que devuelve. La espera de una tarea cancelada devuelve una `OperationCanceledException`. Para obtener más información sobre cómo cancelar un proceso asincrónico, vea Ajustar una aplicación asincrónica (C# y Visual Basic).

Para detectar la excepción, espere la tarea en un bloque `try` y detéctela en el bloque asociado `catch`. Para obtener un ejemplo, vea la sección Ejemplo de método async.

Una tarea puede encontrarse en un estado de error debido a que ocurrieron varias excepciones en el método asincrónico esperado. Por ejemplo, la tarea podría ser el resultado de una llamada a Task.WhenAll. Cuando espera una tarea de este tipo, solo se captura una de las excepciones y no puede predecir qué excepción se capturará. Para obtener un ejemplo, vea la sección Ejemplo de Task.WhenAll.

Ejemplo

En el ejemplo siguiente, el bloque `try` contiene una llamada al método `ProcessString` que puede causar una excepción. La cláusula `catch` contiene el controlador de excepciones que muestra un mensaje en la pantalla. Cuando la instrucción `throw` se llama desde dentro `MyMethod`, el sistema busca la instrucción `catch` y muestra el mensaje `Exception caught`.

```
class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
   at TryFinallyTest.Main() Exception caught.
 * */
```

Ejemplo de dos bloques catch

En el ejemplo siguiente, se utilizan dos bloques de detección y la excepción más específica, que es la que aparece primero, es la que se captura.

Para capturar la excepción menos específica, puede sustituir la instrucción throw en `ProcessString` por la siguiente instrucción: `throw new Exception()`.

Si coloca primero el bloque catch menos específico en el ejemplo, aparece el siguiente mensaje de error: `A previous catch clause already catches all exceptions of this or a super type ('System.Exception')`.

```
class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
 Output:
 System.ArgumentNullException: Value cannot be null.
 at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/
```

Ejemplo de método async

En el ejemplo siguiente se muestra el control de excepciones de los métodos asincrónicos. Para capturar una excepción que produce una tarea asincrónica, coloque la expresión `await` en un bloque `try` y capture la excepción en un bloque `catch`.

Quite la marca de comentario de la línea `throw new Exception` en el ejemplo para demostrar el control de excepciones. La propiedad de la tarea `IsFaulted` se establece en `True`, la propiedad de la tarea `Exception.InnerException` se establece en la excepción, y la excepción se captura en el bloque `catch`.

Quite la marca de comentario de la línea `throw new OperationCanceledException` para ver lo que pasa cuando se cancela un proceso asincrónico. La propiedad de la tarea `IsCanceled` se establece en `true`, y la excepción se captura en el bloque `catch`. En algunas condiciones que no son aplicables a este ejemplo, la propiedad de la tarea `IsFaulted` se establece en `true` y `IsCanceled` se establece en `false`.

```
public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted:  " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted:  False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted:  True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted:  False
```

Ejemplo de Task.WhenAll

En el ejemplo siguiente se muestra el control de excepciones en el que varias tareas pueden producir varias excepciones. El bloque `try` espera la tarea devuelta por una llamada a Task.WhenAll. La tarea se completa cuando se hayan completado las tres tareas a las que se aplica el método WhenAll.

Cada una de las tres tareas produce una excepción. El bloque `catch` se itera a través de las excepciones, que se encuentran en la propiedad `Exception.InnerExceptions` de la tarea devuelta por Task.WhenAll.

```
public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerExceptions)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task
```

especificación del lenguaje C#

Para obtener más información, vea la sección sobre la instrucción try de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Instrucciones try, throw y catch (C++)
- throw
- try-finally
- Cómo: Iniciar excepciones explícitamente

try-finally (Referencia de C#)

18/03/2020 • 5 minutes to read • Edit Online

Mediante el uso de un bloque `finally`, puede limpiar todos los recursos asignados en un bloque try y ejecutar código incluso si se produce una excepción en el bloque `try`. Normalmente, las instrucciones de un bloque `finally` se ejecutan cuando el control abandona una instrucción `try`. La transferencia de control se puede producir como resultado de la ejecución normal, de la ejecución de una instrucción `break`, `continue`, `goto` o `return`, o de la propagación de una excepción fuera de la instrucción `try`.

Dentro de una excepción controlada, se garantiza la ejecución del bloque `finally` asociado. Pero en el caso de una excepción no controlada, la ejecución del bloque `finally` depende de la manera en que se desencadene la operación de desenredo de la excepción. Esto, a su vez, depende de cómo esté configurado el equipo.

Normalmente, cuando una excepción no controlada finaliza una aplicación, no es importante si el bloque `finally` se ejecuta o no. Pero si tiene instrucciones en un bloque `finally` que debe ejecutarse incluso en esa situación, una solución es agregar un bloque `catch` a la instrucción `try` - `finally`. Como alternativa, puede capturar la excepción que se podría producir en el bloque `try` de una instrucción `try` - `finally` más arriba en la pila de llamadas. Es decir, puede capturar la excepción en el método que llama al método que contiene la instrucción `try` - `finally`, en el método que llama a ese método o en cualquier método en la pila de llamadas. Si no se captura la excepción, la ejecución del bloque `finally` depende de si el sistema operativo decide desencadenar una operación de desenredo de la excepción.

Ejemplo

En el ejemplo siguiente, una instrucción de conversión no válida provoca una excepción `System.InvalidCastException`. Se trata de una excepción no controlada.

```
public class ThrowTestA
{
    static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is triggered.
    // i = 123
}
```

En el ejemplo siguiente, se captura una excepción del método `TryCast` en un método más arriba en la pila de llamadas.

```
public class ThrowTestB
{
    static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine
                ("Catching the {0} exception triggers the finally block.",
                ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    public static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
    // Output:
    // In the finally block in TryCast, i = 123.

    // Catching the System.InvalidCastException exception triggers the finally block.

    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}
```

Para obtener más información sobre `finally`, vea try-catch-finally.

C# también contiene la instrucción using, que proporciona una función similar para objetos IDisposable en una sintaxis adecuada.

especificación del lenguaje C#

Para obtener más información, vea la sección sobre la instrucción try de la Especificación del lenguaje C#.

Vea también

- Referencia de C#

- Guía de programación de C#
- Palabras clave de C#
- Instrucciones try, throw y catch (C++)
- throw
- try-catch
- Cómo: Iniciar excepciones explícitamente

try-catch-finally (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Un uso habitual de `catch` y `finally` juntos es obtener y usar recursos de un bloque `try`, lidiar con circunstancias excepcionales de un bloque `catch` y liberar los recursos del bloque `finally`.

Para más información y ejemplos sobre cómo volver a iniciar excepciones, vea try-catch y Generación de excepciones. Para más información sobre el bloque `finally`, vea try-finally.

Ejemplo

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }

        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

especificación del lenguaje C#

Para obtener más información, vea la sección sobre la instrucción try de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Instrucciones try, throw y catch (C++)
- throw
- Cómo: Iniciar excepciones explícitamente
- using (instrucción)

Checked y Unchecked (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Las instrucciones de C# se pueden ejecutar en un contexto comprobado o no comprobado. En un contexto no comprobado, el desbordamiento aritmético produce una excepción. En un contexto sin comprobar, se omite el desbordamiento aritmético y se produce un truncamiento del resultado al descartar los bits de orden superior que no caben en el tipo de destino.

- checked Especifica un contexto comprobado.
- unchecked Especifica un contexto no comprobado.

Las siguientes operaciones se ven afectadas por la comprobación del desbordamiento:

- Expresiones que usan los siguientes operadores predefinidos en tipos integrales:

  `++`, `--`, `-` unario, `+`, `-`, `*`, `/`

- Conversiones numéricas explícitas entre tipos integrales o de `float` o `double` a un tipo integral.

Si no se especifica `checked` ni `unchecked`, el contexto predeterminado para expresiones no constantes (expresiones que se evalúan en tiempo de ejecución) se define por medio del valor de la opción del compilador /checked. De forma predeterminada, el valor de esa opción se desactiva y se ejecutan operaciones aritméticas en un contexto sin comprobar.

Para expresiones constantes (expresiones que se pueden evaluar completamente en tiempo de compilación), el contexto predeterminado se comprueba siempre. A menos que se coloque de forma explícita una expresión constante en un contexto sin comprobar, los desbordamientos que se producen durante la evaluación de tiempo de compilación de la expresión dan lugar a errores en tiempo de compilación.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Palabras clave de instrucciones

checked (Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

La palabra clave `checked` se usa con el fin de habilitar explícitamente la comprobación de desbordamiento para operaciones aritméticas y conversiones de tipo integral.

De manera predeterminada, una expresión que solo contiene valores constantes provoca un error del compilador si la expresión genera un valor fuera del intervalo del tipo de destino. Si la expresión contiene uno o varios valores no constantes, el compilador no detecta el desbordamiento. En el siguiente ejemplo, la evaluación de la expresión asignada a `i2` no genera un error del compilador.

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

De manera predeterminada, estas expresiones no constantes no se someten a la comprobación de desbordamiento en tiempo de ejecución y no generan excepciones de desbordamiento. En el ejemplo anterior, se muestra -2 147 483 639 como la suma de dos enteros positivos.

La comprobación de desbordamiento se puede habilitar mediante opciones del compilador, la configuración del entorno o la palabra clave `checked`. En los ejemplos siguientes, se muestra cómo usar una expresión `checked` o un bloque `checked` para detectar el desbordamiento generado por la suma anterior en tiempo de ejecución. En ambos ejemplos se genera una excepción de desbordamiento.

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

Se puede usar la palabra clave unchecked para evitar la comprobación de desbordamiento.

Ejemplo

En este ejemplo, se muestra cómo usar `checked` para habilitar la comprobación de desbordamiento en tiempo de ejecución.

```
class OverFlowTest
{
    // Set maxIntValue to the maximum value for integers.
    static int maxIntValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxIntValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT:  " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxIntValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT:  " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
                          CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
                          UncheckedMethod());
    }
    /*
   Output:
   CHECKED and CAUGHT:  System.OverflowException: Arithmetic operation resulted
   in an overflow.
      at ConsoleApplication1.OverFlowTest.CheckedMethod()

   CHECKED output value is: 0
   UNCHECKED output value is: -2147483639
 */
}
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Checked y Unchecked
- unchecked

unchecked (Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

La palabra clave `unchecked` se usa para suprimir la comprobación de desbordamiento en las operaciones y conversiones aritméticas de tipos enteros.

En un contexto unchecked, si una expresión genera un valor que está fuera del intervalo del tipo de destino, no se marca el desbordamiento. Por ejemplo, dado que el cálculo del siguiente ejemplo se realiza en un bloque o una expresión `unchecked`, el hecho de que el resultado sea demasiado grande para un entero se omite y se asigna a `int1` el valor -2 147 483 639.

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Si se quita el entorno `unchecked`, se produce un error de compilación. El desbordamiento se puede detectar en tiempo de compilación porque todos los términos de la expresión son constantes.

Las expresiones que contienen términos no constantes son unchecked de forma predeterminada en tiempo de compilación y tiempo de ejecución. Vea checked para obtener información sobre cómo habilitar un entorno checked.

Puesto que la comprobación de desbordamiento lleva tiempo, el uso del código unchecked en situaciones donde no hay riesgo de desbordamiento puede mejorar el rendimiento. Pero si el desbordamiento es una posibilidad, se debe usar un entorno checked.

Ejemplo

En este ejemplo se muestra cómo usar la palabra clave `unchecked`.

```
class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);

        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la

fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Checked y Unchecked
- checked

fixed (Instrucción, Referencia de C#)

29/06/2020 • 5 minutes to read • Edit Online

La instrucción `fixed` evita que el recolector de elementos no utilizados reubique una variable móvil. La instrucción `fixed` solo se permite en un contexto de unsafe. También puede usar la palabra clave `fixed` para crear búferes de tamaño fijo.

La instrucción `fixed` establece un puntero a una variable administrada y "ancla" esa variable durante su ejecución. Los punteros a variables administradas móviles solo son útiles en un contexto `fixed`. Sin un contexto `fixed`, la recolección de elementos no utilizados podría reubicar las variables de forma impredecible. El compilador de C# solo permite asignar un puntero a una variable administrada en una instrucción `fixed`.

```
class Point
{
    public int x;
    public int y;
}

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

Puede inicializar un puntero mediante una matriz, una cadena, un búfer de tamaño fijo o la dirección de una variable. En el ejemplo siguiente se muestra el uso de direcciones, matrices y cadenas de variable:

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

A partir de C# 7.3, la instrucción `fixed` funciona en tipos adicionales más allá de matrices, cadenas, búferes de tamaño fijo o variables no administradas. Cualquier tipo que implemente un método denominado `GetPinnableReference` se puede anclar. `GetPinnableReference` debe devolver una variable `ref` a un tipo no administrado. Los tipos de .NET System.Span<T> y System.ReadOnlySpan<T> presentados en .NET Core 2.0 usan este patrón y se pueden anclar. Esto se muestra en el ejemplo siguiente:

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
                  1,
                1,  1,
              1,  2,  1,
            1,  3,  3,  1,
          1,  4,  6,  4,  1,
        1,  5,  10, 10, 5,  1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

Si crea tipos que deben participar en este patrón, consulte Span<T>.GetPinnableReference() para ver un ejemplo de implementación del patrón.

Es posible inicializar varios punteros en una sola instrucción si todos son del mismo tipo:

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

Para inicializar punteros de tipos diferentes, simplemente anide instrucciones `fixed`, como se muestra en el ejemplo siguiente.

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

Después de ejecutar el código de la instrucción, las variables ancladas se desanclan y quedan sujetas a la recolección de elementos no utilizados. Por lo tanto, no debe apuntar a esas variables fuera de la instrucción `fixed`. Las variables declaradas en la instrucción `fixed` se limitan a dicha instrucción, lo que simplifica esta tarea:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
   ...
}
// ps and pd are no longer in scope here.
```

Los punteros inicializados en instrucciones `fixed` son variables de solo lectura. Si quiere modificar el valor del puntero, debe declarar una segunda variable de puntero y modificarla. La variable declarada en la instrucción `fixed` no se puede modificar:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

Puede asignar memoria en la pila, donde no está sujeta a la recolección de elementos no utilizados y, por tanto, no necesita anclarse. Para ello, use una expresión `stackalloc`.

Especificación del lenguaje C#

Para más información, consulte la sección La declaración fixed de Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- unsafe
- Tipos de puntero
- Búferes de tamaño fijo

Instrucción lock (Referencia de C#)

29/06/2020 • 4 minutes to read • Edit Online

La instrucción `lock` adquiere el bloqueo de exclusión mutua de un objeto determinado, ejecuta un bloque de instrucciones y luego libera el bloqueo. Mientras se mantiene un bloqueo, el subproceso que lo mantiene puede volver a adquirir y liberar el bloqueo. Ningún otro subproceso puede adquirir el bloqueo y espera hasta que se libera.

La instrucción `lock` tiene el formato

```
lock (x)
{
    // Your code...
}
```

donde `x` es una expresión de un tipo de referencia. Es exactamente equivalente a

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Puesto que el código usa un bloque try... finally, el bloqueo se libera aunque se produzca una excepción dentro del cuerpo de una instrucción `lock`.

No se puede usar el operador await en el cuerpo de una instrucción `lock`.

Instrucciones

Al sincronizar el acceso del subproceso al recurso compartido, bloquee una instancia dedicada de objeto (por ejemplo, `private readonly object balanceLock = new object();`) u otra instancia cuyo empleo como objeto de bloqueo sea poco probable por parte de elementos no relacionados del código. Evite el uso de la misma instancia de objeto de bloqueo para distintos recursos compartidos, ya que se podría producir un interbloqueo o una contención de bloqueo. En particular, evite utilizar lo siguiente como objetos de bloqueo:

- `this`, porque los autores de llamadas podrían usarlo como un bloqueo.
- Instancias Type, porque el operador o la reflexión typeof podrían obtenerlas.
- Instancias de cadena, incluidos literales de cadena, porque podrían internarse.

Mantenga el bloqueo durante el menor tiempo posible para reducir la contención de bloqueo.

Ejemplo

En el ejemplo siguiente se define una clase `Account` que sincroniza el acceso a su campo privado `balance` mediante el bloqueo de una instancia dedicada `balanceLock`. El empleo de la misma instancia para bloquear

garantiza que el campo `balance` no sea actualizado al mismo tiempo por dos subprocesos que intentan llamar a los métodos `Debit` o `Credit` simultáneamente.

```
using System;
using System.Threading.Tasks;

public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
```

```
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
        foreach (var amount in amounts)
        {
            if (amount >= 0)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(Math.Abs(amount));
            }
        }
    }
}
```

Especificación del lenguaje C#

Para más información, consulte la sección sobre la instrucción lock de la especificación del lenguaje C#.

Vea también

- Referencia de C#
- Palabras clave de C#
- System.Threading.Monitor
- System.Threading.SpinLock
- System.Threading.Interlocked
- Información general sobre las primitivas de sincronización

Parámetros de métodos (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Los parámetros declarados para un método sin in, ref o out se pasan al método llamado por valor. Ese valor se puede cambiar en el método, pero el cambio se perderá cuando se devuelva el control al procedimiento que ha realizado la llamada. Si usa palabras clave de parámetros de método en la declaración del parámetro, puede modificar este comportamiento.

Esta sección describe las palabras clave que puede usar para declarar parámetros de métodos:

- params especifica que este parámetro puede tomar un número variable de argumentos.
- in especifica que este parámetro se pasa por referencia, pero solo se lee mediante el método llamado.
- ref especifica que este parámetro se pasa por referencia y puede ser leído o escrito por el método llamado.
- out especifica que este parámetro se pasa por referencia y se escribe mediante el método llamado.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#

params (Referencia de C#)

04/05/2020 • 2 minutes to read • Edit Online

Mediante el uso de la palabra clave `params`, puede especificar un parámetro de método que toma un número variable de argumentos. El tipo de parámetro debe ser una matriz unidimensional.

No se permiten parámetros adicionales después de la palabra clave `params` en una declaración de método, y solo se permite una palabra clave `params` en una declaración de método.

Si el tipo declarado del parámetro `params` no es una matriz unidimensional, se produce el error CS0225 del compilador.

Cuando se llama a un método con un parámetro `params`, se puede pasar:

- Una lista separada por comas de argumentos del tipo de los elementos de la matriz.
- Una matriz de argumentos del tipo especificado.
- Sin argumentos. Si no envía ningún argumento, la longitud de la lista `params` es cero.

Ejemplo

En el ejemplo siguiente se muestran varias maneras de enviar argumentos a un parámetro `params`.

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Parámetros de métodos

Modificador del parámetro in (referencia de C#)

02/04/2020 • 10 minutes to read • Edit Online

La palabra clave `in` hace que los argumentos se pasen por referencia. Hace que el parámetro formal sea un alias para el argumento, que debe ser una variable. En otras palabras, cualquier operación en el parámetro se realiza en el argumento. Es como las palabras clave ref o out, salvo que el método al que se llama no puede modificar los argumentos `in`. Mientras que los argumentos `ref` se pueden modificar, el método llamado debe modificar los argumentos `out` y esas modificaciones se pueden observar en el contexto de la llamada.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

En el ejemplo anterior se muestra que el modificador `in` no suele ser necesario en el sitio de llamada, sino que solo lo es en la declaración del método.

> **NOTE**
>
> Además, la palabra clave `in` puede usarse con un parámetro de tipo genérico para especificar que el parámetro de tipo es contravariante, parte de una instrucción `foreach` o de una cláusula `join` de una consulta de LINQ. Para más información sobre el uso de la palabra clave `in` en esos contextos, vea in, que además incluye vínculos a todos estos usos.

Las variables que se han pasado como argumentos `in` deben inicializarse antes de pasarse en una llamada de método. Sin embargo, es posible que el método llamado no asigne ningún valor o modifique el argumento.

El modificador de parámetro `in` está disponible en C# 7.2 y versiones posteriores. Las versiones anteriores generan el error del compilador `CS8107` ("Feature 'readonly references' is not available in C# 7.0. Please use language version 7.2 or greater.") (La característica "readonly references" no está disponible en C# 7.0. Use la versión de lenguaje 7.2 o superior"). Para configurar la versión del lenguaje de compilador, vea Seleccionar la versión del lenguaje C#.

Las palabras clave `in`, `ref` y `out` no se consideran parte de la firma del método con el fin de resolver la sobrecarga. Por lo tanto, los métodos no pueden sobrecargarse si la única diferencia es que un método toma un argumento `ref` o `in` y el otro toma un argumento `out`. Por ejemplo, el código siguiente, no se compilará:

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

Está permitida la sobrecarga en función de la presencia de `in`:

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

Reglas de resolución de sobrecarga

Puede comprender las reglas de resolución de sobrecarga de métodos por valor frente a argumentos `in` mediante la comprensión de la motivación de argumentos `in`. Definir métodos usando parámetros `in` es una optimización del rendimiento potencial. Algunos argumentos de tipo `struct` pueden tener un gran tamaño y, cuando se llama a métodos en bucles de pequeñas dimensiones o rutas de acceso de código crítico, el costo de copiar esas estructuras resulta crucial. Los métodos declaran parámetros `in` para especificar qué argumentos pueden pasarse por referencia sin ningún riesgo porque el método llamado no modifica el estado de ese argumento. Al pasar esos argumentos por referencia se evita la copia (potencialmente) costosa.

Especificar `in` en argumentos en el sitio de llamada suele ser opcional. No hay ninguna diferencia semántica entre pasar argumentos por valor y pasarlos por referencia mediante el modificador `in`. El modificador `in` en el sitio de llamada es opcional porque no es necesario indicar que podría cambiarse el valor del argumento. Se agrega explícitamente el modificador `in` en el sitio de llamada para garantizar que el argumento se pasa por referencia, y no por valor. Usar explícitamente `in` tiene los dos efectos siguientes:

El primero es que, al especificar `in` en el sitio de llamada, se fuerza al compilador a seleccionar un método definido con un parámetro `in` coincidente. En caso contrario, cuando dos métodos se diferencian solo en presencia de `in`, la sobrecarga por valor es una coincidencia mejor.

El segundo es que, al especificar `in` declara su intención para pasar un argumento por referencia. Los argumentos usados con `in` deben representar una ubicación a la que se pueda hacer referencia directamente. Se aplican las mismas reglas generales para los argumentos `out` y `ref`: no se pueden usar constantes, propiedades normales u otras expresiones que produzcan valores. En caso contrario, si se omite `in` en el sitio de llamada, se informa al compilador de que le permitirá crear una variable temporal para pasar por referencia de solo lectura para el método. El compilador crea una variable temporal para superar varias restricciones con argumentos `in`:

- Una variable temporal permite constantes en tiempo de compilación como parámetros `in`.
- Una variable temporal permite propiedades u otras expresiones para parámetros `in`.
- Una variable temporal permite argumentos en los que hay una conversión implícita desde el tipo de argumento hacia el tipo de parámetro.

En todas las instancias anteriores, el compilador crea una variable temporal que almacena el valor de la constante, la propiedad u otra expresión.

Estas reglas se muestran en este código:

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Supongamos ahora que hay disponible otro método que usa argumentos por valor. Los resultados cambian como se muestra en este código:

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

La única llamada de método donde se pasa el argumento por referencia es la última.

> **NOTE**
> El código anterior usa `int` como el tipo de argumento para simplificar el trabajo. Como `int` no es más grande que una referencia en la mayoría de máquinas modernas, no supone ninguna ventaja pasar un único `int` como una referencia de solo lectura.

Limitaciones de los parámetros `in`

Las palabras clave `in`, `ref` y `out` no pueden usarse para estos tipos de métodos:

- Métodos asincrónicos, que se definen mediante el uso del modificador async.
- Métodos de iterador, que incluyen una instrucción yield return o `yield break`.
- El primer argumento de un método de extensión no puede tener el modificador `in` a menos que ese argumento sea una estructura.
- El primer argumento de un método de extensión donde el argumento es un tipo genérico (incluso si el tipo está restringido para ser una estructura).

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Parámetros de métodos
- Escritura de código seguro y eficaz

ref (Referencia de C#)

04/05/2020 • 16 minutes to read • Edit Online

La palabra clave `ref` indica un valor que se ha pasado mediante referencia. Se usa en cuatro contextos diferentes:

- En una firma del método y en una llamada al método, para pasar un argumento a un método mediante referencia. Para más información, vea Pasar un argumento mediante referencia.
- En una firma del método, para devolver un valor al autor de la llamada mediante referencia. Para obtener más información, consulte Valores devueltos de referencia.
- En un cuerpo de miembro, para indicar que un valor devuelto de referencia se almacena localmente como una referencia que el autor de la llamada pretende modificar o, en general, que una variable local accede a otro valor por referencia. Para más información, vea Variables locales de tipo ref.
- En una declaración `struct` para declarar `ref struct` o `readonly ref struct`. Para obtener más información, vea la sección struct `ref` del artículo tipos de estructura.

Pasar un argumento mediante referencia

Cuando se usa en una lista de parámetros del método, la palabra clave `ref` indica que un argumento se ha pasado mediante referencia, no por valor. La palabra clave `ref` hace que el parámetro formal sea un alias para el argumento, que debe ser una variable. En otras palabras, cualquier operación en el parámetro se realiza en el argumento. Por ejemplo, si el autor de la llamada pasa una expresión de variable local o una expresión de acceso de elemento de matriz, y el método llamado reemplaza el objeto al que hace referencia el parámetro ref, entonces la variable local del autor de la llamada o el elemento de matriz hacen ahora referencia al nuevo objeto cuando el método devuelve resultados.

> **NOTE**
>
> No confunda el concepto de pasar por referencia con el concepto de tipos de referencia. Estos dos conceptos no son lo mismo. Un parámetro de método puede ser modificado por `ref` independientemente de si se trata de un tipo de valor o de un tipo de referencia. No hay ninguna conversión boxing de un tipo de valor cuando se pasa por referencia.

Para usar un parámetro `ref`, la definición de método y el método de llamada deben utilizar explícitamente la palabra clave `ref`, como se muestra en el ejemplo siguiente.

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

Un argumento que se pasa a un parámetro `ref` o `in` debe inicializarse antes de pasarlo. En esto difiere de los parámetros out, cuyos argumentos no tienen que inicializarse explícitamente antes de pasarlos.

Los miembros de una clase no pueden tener signaturas que se diferencien solo por `ref`, `in` o `out`. Si la única diferencia entre dos miembros de un tipo es que uno de ellos tiene un parámetro `ref` y el otro tiene un parámetro `out` o `in`, se produce un error de compilador. El código siguiente, por ejemplo, no se compila.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

En cambio, los métodos pueden sobrecargarse cuando un método tiene un parámetro `ref`, `in` o `out` y el otro tiene un parámetro de valor, como se muestra en el ejemplo siguiente.

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

En otras situaciones que requieran firma coincidente, como ocultar o reemplazar, `in`, `ref` y `out` forman parte de la signatura y no coinciden entre sí.

Las propiedades no son variables. Son métodos y no se pueden pasar a parámetros `ref`.

Las palabras clave `ref`, `in` y `out` no pueden usarse para estos tipos de métodos:

- Métodos asincrónicos, que se definen mediante el uso del modificador async.
- Métodos de iterador, que incluyen una instrucción yield return o `yield break`.

Además, los métodos de extensión tienen las restricciones siguientes:

- No se puede usar la palabra clave `out` en el primer argumento de un método de extensión.
- No se puede usar la palabra clave `ref` en el primer argumento de un método de extensión cuando el argumento no es un struct ni un tipo genérico no restringido a ser un struct.
- No se puede usar la palabra clave `in` a menos que el primer argumento sea un struct. No se puede usar la palabra clave `in` en ningún tipo genérico, incluso cuando está restringido a ser un struct.

Paso de un argumento mediante referencia: Un ejemplo

En los ejemplos anteriores se pasan tipos de valor mediante referencia. También se puede usar la palabra clave `ref` para pasar tipos de referencia mediante referencia. Pasar un tipo de referencia por referencia permite que el método llamado pueda reemplazar el objeto al que hace referencia el parámetro de referencia en el autor de la llamada. La ubicación de almacenamiento del objeto se pasa al método como el valor del parámetro de referencia. Si cambia el valor de la ubicación de almacenamiento del parámetro (para que apunte a un nuevo objeto), también debe cambiar la ubicación de almacenamiento a la que se refiere el autor de la llamada. En el ejemplo siguiente se pasa una instancia de un tipo de referencia como un parámetro `ref`.

```
class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main.  Name: Fasteners, ID: 54321
// Back in Main.  Name: Stapler, ID: 12345
```

Para obtener más información sobre cómo pasar tipos de referencia por valor y por referencia, vea Pasar parámetros Reference-Type .

Valores devueltos de referencia

Los valores devueltos de referencia (o valores devueltos de tipo ref) son valores que devuelve un método mediante referencia al autor de la llamada. Es decir, el autor de la llamada puede modificar el valor devuelto por un método, y ese cambio se refleja en el estado del objeto del método de llamada.

Un valor devuelto de referencia se define mediante la palabra clave `ref`:

- En la firma del método. Por ejemplo, en la firma de método siguiente se indica que el método `GetCurrentPrice` devuelve un valor Decimal por referencia.

```
public ref decimal GetCurrentPrice()
```

- Entre el token `return` y la variable devuelta en una instrucción `return` en el método. Por ejemplo:

```
return ref DecimalArray[0];
```

Para que el autor de la llamada modifique el estado del objeto, el valor devuelto de referencia debe almacenarse en una variable que se defina explícitamente como una variable local de tipo ref.

Este es un ejemplo de valor devuelto de referencia más completo que muestra la firma y el cuerpo del método.

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

El método llamado también puede declarar el valor devuelto como `ref readonly` para devolver el valor por referencia y exigir que el código de llamada no pueda modificar el valor devuelto. El método de llamada puede evitar copiar el valor devuelto mediante el almacenamiento del valor en una variable de tipo ref readonly.

Para obtener un ejemplo, vea Un ejemplo de valores devueltos y variables locales de tipo ref.

Variables locales de tipo ref

Una variable local de tipo ref se usa para hacer referencia a valores devueltos con `return ref`. Una variable local de tipo ref no se puede inicializar en un valor devuelto de tipo no ref. Es decir, el lado derecho de la inicialización debe ser una referencia. Cualquier modificación en el valor de la variable local de tipo ref se refleja en el estado del objeto cuyo método ha devuelto el valor mediante referencia.

Defina una variable local de tipo ref mediante la palabra clave `ref` antes de la declaración de variable, así como inmediatamente antes de la llamada al método que devuelve el valor mediante referencia.

Por ejemplo, en la siguiente instrucción se define un valor de variable local de tipo ref que se devuelve mediante un método denominado `GetEstimatedValue`:

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

Puede acceder a un valor por referencia de la misma manera. En algunos casos, acceder a un valor por referencia aumenta el rendimiento, ya que evita una operación de copia potencialmente cara. Por ejemplo, en la instrucción siguiente se muestra cómo es posible definir un valor local de referencia que se usa para hacer referencia a un valor.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

En ambos ejemplos la palabra clave `ref` debe usarse en ambos lugares. De lo contrario, el compilador genera el error CS8172, "No se puede inicializar una variable por referencia con un valor".

A partir C# 7.3, la variable de iteración de la instrucción `foreach` puede ser una variable ref local o ref readonly local. Para más información, vea el artículo sobre la instrucción foreach.

A partir C# 7.3, las variables locales de tipo ref o locales de tipo ref readonly se pueden reasignar con el operador de asignación ref.

Variables locales de tipo ref readonly

Una variable local de tipo ref readonly se usa para hacer referencia a los valores devueltos por el método o la propiedad que tiene `ref readonly` en su firma y utiliza `return ref`. Una variable `ref readonly` combina las propiedades de una `ref` variable local con una variable `readonly`: es un alias para el almacenamiento al que se asigna y no se puede modificar.

Un ejemplo de valores devueltos y variables locales de tipo ref

En el ejemplo siguiente se define una clase `Book` que tiene dos campos String, `Title` y `Author`. También define una clase `BookCollection` que incluye una matriz privada de objetos `Book`. Los objetos book individuales se devuelven mediante referencia llamando a su método `GetBookByTitle`.

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London"
},
                        new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
                       };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

Cuando el autor de la llamada almacena el valor devuelto mediante el método `GetBookByTitle` como una variable local de tipo ref, los cambios que el autor de la llamada realiza en el valor devuelto se reflejan en el objeto `BookCollection`, como se muestra en el ejemplo siguiente.

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//       Call of the Wild, The, by Jack London
//       Tale of Two Cities, A, by Charles Dickens
//
//       Republic, The, by Plato
//       Tale of Two Cities, A, by Charles Dickens
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Escritura de código seguro y eficaz
- Devoluciones y variables locales ref
- Expresión condicional ref
- Pasar parámetros
- Parámetros de métodos
- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#

Modificador del parámetro out (Referencia de C#)

16/04/2020 • 7 minutes to read • Edit Online

La palabra clave `out` hace que los argumentos se pasen por referencia. Hace que el parámetro formal sea un alias para el argumento, que debe ser una variable. En otras palabras, cualquier operación en el parámetro se realiza en el argumento. Esto es como la palabra clave ref, salvo que `ref` requiere que se inicialice la variable antes de pasarla. También es como la palabra clave in, salvo que `in` no permite que el método llamado modifique el valor del argumento. Para usar un parámetro `out`, tanto la definición de método como el método de llamada deben utilizar explícitamente la palabra clave `out`. Por ejemplo:

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

> **NOTE**
>
> La palabra clave `out` también puede usarse con un parámetro de tipo genérico para especificar que el parámetro de tipo es covariante. Para obtener más información sobre el uso de la palabra clave `out` en este contexto, vea Out (Modificador genérico).

Las variables que se han pasado como argumentos `out` no tienen que inicializarse antes de pasarse en una llamada al método. En cambio, se necesita el método que se ha llamado para asignar un valor antes de que el método se devuelva.

Las palabras clave `in`, `ref` y `out` no se consideran parte de la firma del método con el fin de resolver la sobrecarga. Por lo tanto, los métodos no pueden sobrecargarse si la única diferencia es que un método toma un argumento `ref` o `in` y el otro toma un argumento `out`. Por ejemplo, el código siguiente, no se compilará:

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

En cambio, la sobrecarga es legal si un método toma un argumento `ref`, `in` o `out` y el otro no tiene ninguno de estos modificadores, como se muestra aquí:

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

El compilador elige la mejor sobrecarga haciendo coincidir los modificadores de parámetro del sitio de llamada con los usados en la llamada de método.

Las propiedades no son variables y, por tanto, no pueden pasarse como parámetros `out`.

Las palabras clave `in`, `ref` y `out` no pueden usarse para estos tipos de métodos:

- Métodos asincrónicos, que se definen mediante el uso del modificador async.
- Métodos de iterador, que incluyen una instrucción yield return o `yield break`.

Además, los métodos de extensión tienen las restricciones siguientes:

- No se puede usar la palabra clave `out` en el primer argumento de un método de extensión.
- No se puede usar la palabra clave `ref` en el primer argumento de un método de extensión cuando el argumento no es un struct ni un tipo genérico no restringido a ser un struct.
- No se puede usar la palabra clave `in` a menos que el primer argumento sea un struct. No se puede usar la palabra clave `in` en ningún tipo genérico, incluso cuando está restringido a ser un struct.

Declaración de parámetros `out`

Declarar un método con el argumento `out` es una solución alternativa clásica para devolver varios valores. A partir de C# 7.0, considere las tuplas para escenarios similares. En el ejemplo siguiente, se utiliza `out` para devolver tres variables con una única llamada al método. El tercer argumento se asigna a NULL. Esto permite que los métodos devuelvan valores opcionalmente.

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

Llamar a un método con un argumento `out`

En C# 6 y versiones anteriores, debe declarar una variable en una instrucción independiente antes de pasarla como un argumento `out`. En el ejemplo siguiente se declara una variable denominada

`number` antes de que se pase al método Int32.TryParse, que intenta convertir una cadena en un número.

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

A partir de C# 7.0, puede declarar la variable `out` en la lista de argumentos de la llamada al método, en lugar de en una declaración de variable independiente. Esto genera un código legible más compacto y, además, evita que asigne un valor a la variable antes de la llamada al método de manera involuntaria. El ejemplo siguiente es como el ejemplo anterior, excepto que define la variable `number` en la llamada al método Int32.TryParse.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

En el ejemplo anterior, la variable `number` está fuertemente tipada como `int`. También puede declarar una variable local con tipo implícito como se muestra en el siguiente ejemplo.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Parámetros de métodos

espacio de nombres (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `namespace` se usa para declarar un ámbito que contiene un conjunto de objetos relacionados. Puede usar un espacio de nombres para organizar los elementos de código y crear tipos únicos globales.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

Comentarios

En un espacio de nombres, se pueden declarar cero o más de los siguientes tipos:

- otro espacio de nombres
- class
- interface
- struct
- enum
- delegate

Tanto si se declara explícitamente un espacio de nombres en un archivo de código fuente de C# como si no, el compilador agrega un espacio de nombres predeterminado. Este espacio de nombres sin nombre, que a veces se denomina espacio de nombres global, está presente en todos los archivos. Todos los identificadores del espacio de nombres global están disponibles para su uso en un espacio de nombres con nombre.

Los espacios de nombres tienen implícitamente acceso público y esto no se puede modificar. Para ver una descripción de los modificadores de acceso que se pueden asignar a los elementos de un espacio de nombres, vea Modificadores de acceso.

Es posible definir un espacio de nombres en dos o más declaraciones. Por ejemplo, en el ejemplo siguiente se definen dos clases como parte del espacio de nombres `MyCompany`:

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

Ejemplo

En el ejemplo siguiente se muestra cómo llamar a un método estático en un espacio de nombres anidado.

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

especificación del lenguaje C#

Para más información, vea la sección Espacio de nombres de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Palabras clave de C#
- using
- using static
- Calificadores de alias de espacio de nombres `::`
- Espacios de nombres

using (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `using` tiene tres usos principales:

- La instrucción using define un ámbito al final del cual se eliminará un objeto.
- La directiva using crea un alias para un espacio de nombres o importa tipos definidos en otros espacios de nombres.
- La directiva estática using importa los miembros de una sola clase.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Espacios de nombres
- extern

using (Directiva, Referencia de C# Reference)

18/03/2020 • 5 minutes to read • Edit Online

La directiva `using` tiene tres usos:

- Permitir el uso de tipos en un espacio de nombres de manera que no tenga que calificar el uso de un tipo en dicho espacio de nombres:

  ```
  using System.Text;
  ```

- Permitirle acceder a los miembros estáticos y a los tipos anidados de un tipo sin tener que calificar el acceso con el nombre del tipo.

  ```
  using static System.Math;
  ```

  Para obtener más información, vea using static (Directiva).

- Para crear un alias para un espacio de nombre o un tipo. Esto se conoce como *alias using (Directiva)* .

  ```
  using Project = PC.MyCompany.Project;
  ```

La palabra clave `using` también se usa para crear *instrucciones using*, que ayudan a garantizar que los objetos IDisposable, como archivos y fuentes, se tratan correctamente. Consulte using (Instrucción) para obtener más información.

Uso de tipos estáticos

Puede acceder a los miembros estáticos de un tipo sin tener que calificar el acceso con el nombre del tipo:

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Comentarios

El ámbito de una directiva `using` se limita al archivo en el que aparece.

La directiva `using` puede aparecer:

- Al principio de un archivo de código fuente, antes de las definiciones de espacio de nombres o tipo.
- En cualquier espacio de nombres, pero antes de cualquier espacio de nombres o tipos declarados en este espacio de nombres.

De lo contrario, se generará el error de compilador CS1529.

Cree una directiva de alias `using` para facilitar la calificación de un identificador como espacio de nombres o tipo. En cualquier directiva `using`, hay que usar el espacio de nombres o el tipo con cualificación completa, independientemente de las directivas `using` que los precedan. No se puede usar ningún alias `using` en la declaración de una directiva `using`. Por ejemplo, el código siguiente genera un error de compilador:

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

Cree una directiva `using` para usar los tipos de un espacio de nombres sin tener que especificarlo. Una directiva `using` no proporciona acceso a los espacios de nombres que están anidados en el espacio de nombres especificado.

Los espacios de nombres se dividen en dos categorías: definidos por el sistema y definidos por el usuario. Los espacios de nombres definidos por el usuario son espacios de nombres definidos en el código. Para obtener una lista de los espacios de nombres definidos por el sistema, vea Explorador de API de .NET.

Ejemplo 1

En el ejemplo siguiente se muestra cómo definir y usar un alias `using` para un espacio de nombres:

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

Una directiva de alias using no puede tener un tipo genérico abierto en el lado derecho. Por ejemplo, no puede crear un alias using para un elemento `List<T>`, pero puede crear uno para un elemento `List<int>`.

Ejemplo 2

En el ejemplo siguiente se muestra cómo definir una directiva `using` y un alias `using` para una clase:

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

especificación del lenguaje C#

Para obtener más información, vea la sección Directivas using de la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Utilizar espacios de nombres
- Palabras clave de C#
- Espacios de nombres
- using (instrucción)

using static (Directiva, Referencia de C#)

29/06/2020 • 6 minutes to read • Edit Online

La directiva `using static` designa un tipo a cuyos miembros estáticos y tipos anidados se puede acceder sin especificar un nombre de tipo. Su sintaxis es:

```
using static <fully-qualified-type-name>;
```

donde *nombre-completo-del-tipo* es el nombre del tipo a cuyos miembros estáticos y tipos anidados se puede hacer referencia sin especificar un nombre de tipo. Si no proporciona un nombre de tipo completo (el nombre del espacio de nombres completo junto con el nombre del tipo), C# genera el error del compilador CS0246: "No se pudo encontrar el tipo o el nombre del espacio de nombres 'type/namespace' (¿falta una directiva using o una referencia de ensamblado?)".

La directiva `using static` se aplica a cualquier tipo que tenga miembros estáticos (o tipos anidados), aunque también tenga miembros de instancia. Pero los miembros de instancia solo pueden invocarse a través de la instancia del tipo.

La directiva `using static` se ha agregado en C# 6.

Comentarios

Normalmente, cuando se llama a un miembro estático, se especifica el nombre de tipo junto con el nombre de miembro. Especificar varias veces el mismo nombre de tipo para invocar los miembros del tipo puede traducirse en código detallado y poco claro. Por ejemplo, la siguiente definición de una clase `Circle` hace referencia a un número de miembros de la clase Math.

```
using System;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * Math.PI; }
   }

   public double Area
   {
      get { return Math.PI * Math.Pow(Radius, 2); }
   }
}
```

Al eliminar la necesidad de hacer referencia explícitamente a la clase Math cada vez que se hace referencia a un miembro, la directiva `using static` genera un código mucho más limpio:

```
using System;
using static System.Math;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
```

`using static` importa solo los miembros estáticos accesibles y los tipos anidados declarados en el tipo especificado. Los miembros heredados no se importan. Puede importar desde cualquier tipo con nombre con una directiva estática using, incluidos los módulos de Visual Basic. Las funciones de nivel superior F# pueden importarse si aparecen en los metadatos como miembros estáticos de un tipo con nombre cuyo nombre es un identificador de C# válido.

`using static` habilita los métodos de extensión declarados en el tipo especificado estén para la búsqueda de métodos de extensión. Pero los nombres de los métodos de extensión no se importan en el ámbito de referencia sin calificar del código.

Los métodos con el mismo nombre que se importen desde tipos distintos con distintas directivas `using static` en la misma unidad de compilación o espacio de nombres forman un grupo de métodos. La resolución de sobrecarga dentro de estos grupos de método sigue reglas normales de C#.

Ejemplo

En el ejemplo siguiente se usa la directiva `using static` para que los miembros estáticos de las clases Console, Math y String estén disponibles sin tener que especificar su nombre de tipo.

```
using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
   static void Main()
   {
      Write("Enter a circle's radius: ");
      var input = ReadLine();
      if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
         var c = new Circle(radius);

         string s = "\nInformation about the circle:\n";
         s = s + Format("   Radius: {0:N2}\n", c.Radius);
         s = s + Format("   Diameter: {0:N2}\n", c.Diameter);
         s = s + Format("   Circumference: {0:N2}\n", c.Circumference);
         s = s + Format("   Area: {0:N2}\n", c.Area);
         WriteLine(s);
      }
      else {
         WriteLine("Invalid input...");
      }
   }
}

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
// The example displays the following output:
//       Enter a circle's radius: 12.45
//
//       Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95
```

En el ejemplo, también podría haberse aplicado la directiva `using static` al tipo Double. Esto habría permitido llamar al método TryParse(String, Double) sin especificar un nombre de tipo. Sin embargo, el código que se crea es menos legible, ya que es necesario comprobar las directivas `using static` para determinar el método `TryParse` del tipo numérico al que se llama.

Vea también

- using (directiva)
- Referencia de C#
- Palabras clave de C#
- Utilizar espacios de nombres
- Espacios de nombres

using (Instrucción, Referencia de C#)

29/06/2020 • 7 minutes to read • Edit Online

Ofrece una sintaxis adecuada que garantiza el uso correcto de objetos IDisposable. A partir de C# 8.0, la instrucción `using` garantiza el uso correcto de los objetos IAsyncDisposable.

Ejemplo

En el ejemplo siguiente se muestra cómo usar la instrucción `using`.

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using (var reader = new StringReader(manyLines))
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

A partir de C# 8.0, puede usar la siguiente sintaxis alternativa para la instrucción `using` que no requiere llaves:

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using var reader = new StringReader(manyLines);
string? item;
do {
    item = reader.ReadLine();
    Console.WriteLine(item);
} while(item != null);
```

Comentarios

File y Font son ejemplos de tipos administrados que acceden a recursos no administrados (en este caso, identificadores de archivo y contextos de dispositivo). Hay muchos otros tipos de recursos no administrados y tipos de la biblioteca de clases que los encapsulan. Todos estos tipos deben implementar la interfaz IDisposable o la interfaz IAsyncDisposable.

Cuando la duración de un objeto `IDisposable` se limita a un único método, debe declarar y crear instancias del mismo en la instrucción `using`. La instrucción `using` llama al método Dispose del objeto de forma correcta y (cuando se usa tal y como se muestra anteriormente) también hace que el propio objeto salga del ámbito en cuanto se llame a Dispose. Dentro del bloque `using`, el objeto es de solo lectura y no se puede modificar ni reasignar. Si el objeto implementa `IAsyncDisposable` en lugar de `IDisposable`, la instrucción

`using` llama al objeto DisposeAsync y `awaits` al objeto ValueTask devuelto. Para obtener más información sobre IAsyncDisposable, vea Implementación de un método DisposeAsync.

La instrucción `using` asegura que se llama al método Dispose (o DisposeAsync) aunque se produzca una excepción en el bloque `using`. Puede obtener el mismo resultado si coloca el objeto dentro de un bloque `try` y llama a Dispose (o DisposeAsync) en un bloque `finally`; de hecho, es así como el compilador traduce la instrucción `using`. El ejemplo de código anterior se extiende al siguiente código en tiempo de compilación (tenga en cuenta las llaves adicionales para crear el ámbito limitado del objeto):

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

{
    var reader = new StringReader(manyLines);
    try {
        string? item;
        do {
            item = reader.ReadLine();
            Console.WriteLine(item);
        } while(item != null);
    } finally
    {
        reader?.Dispose();
    }
}
```

La nueva sintaxis de la instrucción `using` se traduce en un código similar. Se abre el bloque `try` en el que se declara la variable. El bloque `finally` se agrega al cierre del bloque de inclusión, normalmente, al final de un método.

Consulte el artículo sobre try-finally para obtener más información sobre la instrucción `try` - `finally`.

Se pueden declarar varias instancias de un tipo en una sola instrucción `using`, tal y como se muestra en el ejemplo siguiente. Tenga en cuenta que no se pueden usar variables con tipo implícito (`var`) cuando se declaran varias variables en una sola instrucción:

```
string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using (StringReader left = new StringReader(numbers),
    right = new StringReader(letters))
{
    string? item;
    do {
        item = left.ReadLine();
        Console.Write(item);
        Console.Write("    ");
        item = right.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

También puede combinar varias declaraciones del mismo tipo con la nueva sintaxis introducida con C# 8, tal y como se muestra en el ejemplo siguiente:

```
string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using StringReader left = new StringReader(numbers),
    right = new StringReader(letters);
string? item;
do {
    item = left.ReadLine();
    Console.Write(item);
    Console.Write("    ");
    item = right.ReadLine();
    Console.WriteLine(item);
} while(item != null);
```

Puede crear una instancia del objeto de recurso y luego pasar la variable a la instrucción `using`, pero esto no es un procedimiento recomendado. En este caso, después de que el control abandone el bloque `using` el objeto permanece en el ámbito, pero probablemente ya no tenga acceso a sus recursos no administrados. En otras palabras, ya no se inicializa totalmente. Si intenta usar el objeto fuera del bloque `using`, corre el riesgo de iniciar una excepción. Por este motivo, es mejor crear una instancia del objeto en la instrucción `using` y limitar su ámbito al bloque `using`.

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

var reader = new StringReader(manyLines);
using (reader)
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
// reader is in scope here, but has been disposed
```

Para obtener más información sobre cómo eliminar objetos `IDisposable`, vea Uso de objetos que implementan IDisposable.

Especificación del lenguaje C#

Para obtener más información, vea el apartado Instrucción using en la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#

- Palabras clave de C#
- using (directiva)
- Recolección de elementos no utilizados
- Uso de objetos que implementan IDisposable
- IDisposable interface (Interfaz IDisposable)
- Instrucción using en C# 8.0

alias externo (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Es posible que deba hacer referencia a dos versiones de ensamblados que tienen los mismos nombres de tipo completos. Por ejemplo, es posible que tenga que usar dos o más versiones de un ensamblado en la misma aplicación. Mediante el uso de un alias de ensamblado externo, los espacios de nombres de cada ensamblado pueden ajustarse en espacios de nombres de nivel de raíz denominados por el alias, lo que permite que se usen en el mismo archivo.

> **NOTE**
>
> La palabra clave extern también se usa como un modificador de método, y declara un método escrito en código no administrado.

Para hacer referencia a dos ensamblados con los mismos nombres de tipo completos, debe especificarse un alias en un símbolo del sistema, como sigue:

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

Esto crea los alias externos `GridV1` y `GridV2`. Para usar estos alias desde dentro de un programa, se hace referencia a ellos mediante la palabra clave `extern`. Por ejemplo:

```
extern alias GridV1;
```

```
extern alias GridV2;
```

Cada declaración de alias externo introduce un espacio de nombres de nivel de raíz adicional que es semejante al espacio de nombres global (pero que no se encuentra en su interior). Por tanto, se puede hacer referencia a los tipos de cada ensamblado sin ambigüedad mediante su nombre completo, con raíz en el alias de espacio de nombres adecuado.

En el ejemplo anterior, `GridV1::Grid` sería el control de cuadrícula de `grid.dll`, y `GridV2::Grid` sería el control de cuadrícula de `grid20.dll`.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Operador ::
- -reference (Opciones del compilador de C#)

is (Referencia de C#)

19/03/2020 • 12 minutes to read • Edit Online

El operador `is` comprueba si el resultado de una expresión es compatible con un tipo determinado o, a partir de C# 7.0, prueba una expresión en un patrón. Para obtener información sobre el operador de prueba de tipos `is`, consulte la sección operador is del artículo Operadores de conversión y prueba de tipos.

Coincidencia de patrones con `is`

A partir de C# 7.0, las instrucciones `is` y switch admiten la coincidencia de patrones. La palabra clave `is` admite los patrones siguientes:

- Patrón de tipo, que comprueba si una expresión se puede convertir en un tipo especificado y, en caso afirmativo, la convierte en una variable de ese tipo.
- Patrón de constante, que comprueba si una expresión se evalúa como un valor constante especificado.
- Patrón var, una coincidencia que siempre se realiza correctamente y enlaza el valor de una expresión a una variable local nueva.

Patrón de tipo

Cuando se usa el patrón de tipo para realizar la coincidencia de patrones, `is` comprueba si una expresión se puede convertir en un tipo especificado y, en caso afirmativo, la convierte en una variable de ese tipo. Es una extensión sencilla de la instrucción `is` que habilita la evaluación y la conversión de tipos concisa. La forma general del patrón de tipos `is` es:

```
expr is type varname
```

Donde *expr* es una expresión que se evalúa como una instancia de un tipo, *type* es el nombre del tipo al que se va a convertir el resultado de *expr* y *varname* es el objeto al que se va a convertir el resultado de *expr* si la prueba `is` es `true`.

La expresión `is` es `true` si *expr* no es `null` y se cumple alguna de las siguientes condiciones:

- *expr* es una instancia del mismo tipo que *type*.
- *expr* es una instancia de un tipo que deriva de *type*. En otras palabras, el resultado de *expr* puede convertirse en una instancia de *type*.
- *expr* tiene un tipo en tiempo de compilación que es una clase base de *type* y *expr* tiene un tipo en tiempo de ejecución que es *type* o se deriva de *type*. El *tipo en tiempo de compilación* de una variable es el tipo de la variable tal como se define en su declaración. El *tipo en tiempo de ejecución* de una variable es el tipo de la instancia que se asigna a esa variable.
- *type* es una instancia de un tipo que implementa la interfaz *type*.

A partir de C# 7.1, *expr* puede tener un tipo de tiempo de compilación definido por un parámetro y sus restricciones.

Si *expr* es `true` y `is` se usa con una instrucción `if`, solo se asigna *varname* dentro de la instrucción `if`. El ámbito de *varname* abarca de la expresión `is` al final del bloque que incluye la instrucción `if`. El uso de *varname* en cualquier otra ubicación genera un error en tiempo de compilación por el uso de una variable que

no se ha asignado.

En el ejemplo siguiente se usa el patrón de tipo `is` para proporcionar la implementación de un método IComparable.CompareTo(Object) de tipo.

```
using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        if (o is Employee e)
        {
            return Name.CompareTo(e.Name);
        }
        throw new ArgumentException("o is not an Employee object.");
    }
}
```

Sin coincidencia de patrones, este código podría escribirse del modo siguiente. El uso de la coincidencia de patrones de tipo genera código más compacto y legible al eliminar la necesidad de comprobar si el resultado de una conversión es `null`.

```
using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        var e = o as Employee;
        if (e == null)
        {
           throw new ArgumentException("o is not an Employee object.");
        }
        return Name.CompareTo(e.Name);
    }
}
```

El patrón de tipo `is` también genera código más compacto al determinar el tipo de un tipo de valor. En el ejemplo siguiente se usa el patrón de tipo `is` para determinar si un objeto es una instancia de `Person` o `Dog` antes de mostrar el valor de una propiedad adecuada.

```
using System;

public class Example
{
   public static void Main()
   {
      Object o = new Person("Jane");
      ShowValue(o);

      o = new Dog("Alaskan Malamute");
      ShowValue(o);
   }

   public static void ShowValue(object o)
   {
      if (o is Person p) {
         Console.WriteLine(p.Name);
      }
      else if (o is Dog d) {
         Console.WriteLine(d.Breed);
      }
   }
}

public struct Person
{
   public string Name { get; set; }

   public Person(string name) : this()
   {
      Name = name;
   }
}

public struct Dog
{
   public string Breed { get; set; }

   public Dog(string breedName) : this()
   {
      Breed = breedName;
   }
}
// The example displays the following output:
// Jane
// Alaskan Malamute
```

El código equivalente sin coincidencia de patrones requiere una asignación independiente que incluya una conversión explícita.

```
using System;

public class Example
{
   public static void Main()
   {
      Object o = new Person("Jane");
      ShowValue(o);

      o = new Dog("Alaskan Malamute");
      ShowValue(o);
   }

   public static void ShowValue(object o)
   {
      if (o is Person) {
         Person p = (Person) o;
         Console.WriteLine(p.Name);
      }
      else if (o is Dog) {
         Dog d = (Dog) o;
         Console.WriteLine(d.Breed);
      }
   }
}

public struct Person
{
   public string Name { get; set; }

   public Person(string name) : this()
   {
      Name = name;
   }
}

public struct Dog
{
   public string Breed { get; set; }

   public Dog(string breedName) : this()
   {
      Breed = breedName;
   }
}
// The example displays the following output:
//       Jane
//       Alaskan Malamute
```

Patrón de constante

Al realizar la coincidencia de patrones con el patrón constante, `is` comprueba si una expresión es igual a una constante especificada. En C# 6 y versiones anteriores, la instrucción switch admite el patrón de constante. A partir de C# 7.0, la instrucción `is` también lo admite. Su sintaxis es:

```
   expr is constant
```

donde *expr* es la expresión que se va a evaluar y *constant* es el valor que se va a comprobar. *constant* puede ser cualquiera de las expresiones de constante siguientes:

- Un valor literal.
- El nombre de una variable `const` declarada.

- Una constante de enumeración.

La expresión de constante se evalúa de la siguiente forma:

- Si *expr* y *constant* son tipos enteros, el operador de igualdad de C# determina si la expresión devuelve `true` (es decir, si `expr == constant`).
- De lo contrario, el valor de la expresión se determina mediante una llamada al método estático Object.Equals(expr, constant).

En el ejemplo siguiente se combinan los patrones de tipo y de constante para probar si un objeto es una instancia de `Dice` y, si es así, para determinar si el valor de una tirada de dados es 6.

```
using System;

public class Dice
{
    Random rnd = new Random();
    public Dice()
    {
    }
    public int Roll()
    {
        return rnd.Next(1, 7);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var d1 = new Dice();
        ShowValue(d1);
    }

    private static void ShowValue(object o)
    {
        const int HIGH_ROLL = 6;

        if (o is Dice d && d.Roll() is HIGH_ROLL)
            Console.WriteLine($"The value is {HIGH_ROLL}!");
        else
            Console.WriteLine($"The dice roll is not a {HIGH_ROLL}!");
     }
}
// The example displays output like the following:
//      The value is 6!
```

La búsqueda de `null` puede realizarse con el patrón de constante. La palabra clave `null` es compatible con la instrucción `is`. Su sintaxis es:

```
   expr is null
```

En el ejemplo siguiente se muestra una comparación de comprobaciones `null`:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        object o = null;

        if (o is null)
        {
            Console.WriteLine("o does not have a value");
        }
        else
        {
            Console.WriteLine($"o is {o}");
        }

        int? x = 10;

        if (x is null)
        {
            Console.WriteLine("x does not have a value");
        }
        else
        {
            Console.WriteLine($"x is {x.Value}");
        }

        // 'null' check comparison
        Console.WriteLine($"'is' constant pattern 'null' check result : { o is null }");
        Console.WriteLine($"object.ReferenceEquals 'null' check result : { object.ReferenceEquals(o, null)
}");
        Console.WriteLine($"Equality operator (==) 'null' check result : { o == null }");
    }

    // The example displays the following output:
    // o does not have a value
    // x is 10
    // 'is' constant pattern 'null' check result : True
    // object.ReferenceEquals 'null' check result : True
    // Equality operator (==) 'null' check result : True
}
```

Patrón var

Una coincidencia de patrones con el patrón `var` siempre se realiza correctamente. Su sintaxis es:

```
expr is var varname
```

Donde el valor de *expr* siempre se asigna a una variable local denominada *varname*. *varname* es una variable del mismo tipo que el tipo en tiempo de compilación de *expr*.

Si *expr* se evalúa como `null`, la expresión de `is` produce `true` y asigna `null` a *varname*. El patrón var es uno de los pocos usos de `is` que produce `true` con un valor `null`.

El patrón `var` se puede usar para crear una variable temporal dentro de una expresión booleana, como se muestra en el ejemplo siguiente:

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        int[] testSet = { 100271, 234335, 342439, 999683 };

        var primes = testSet.Where(n => Factor(n).ToList() is var factors
                                    && factors.Count == 2
                                    && factors.Contains(1)
                                    && factors.Contains(n));

        foreach (int prime in primes)
        {
            Console.WriteLine($"Found prime: {prime}");
        }
    }

    static IEnumerable<int> Factor(int number)
    {
        int max = (int)Math.Sqrt(number);
        for (int i = 1; i <= max; i++)
        {
            if (number % i == 0)
            {
                yield return i;
                if (i != number / i)
                {
                    yield return number / i;
                }
            }
        }
    }
}
// The example displays the following output:
//       Found prime: 100271
//       Found prime: 999683
```

En el ejemplo anterior, la variable temporal se usa para almacenar el resultado de una operación costosa. Tras ello, la variable se puede usar varias veces.

Especificación del lenguaje C#

Para más información, consulte la sección del operador is de la Especificación del lenguaje C# y las siguientes propuestas del lenguaje C#:

- Coincidencia de patrones
- Coincidencia de patrones con genéricos

Vea también

- Referencia de C#
- Palabras clave de C#
- Operadores de conversión y prueba de tipos

Restricción new (Referencia de C#)

29/06/2020 • 2 minutes to read • Edit Online

La restricción `new` especifica que un tipo de argumento en una declaración de clase genérica debe tener un constructor sin parámetros público. Para usar la restricción `new`, el tipo no puede ser abstracto.

Aplique la restricción `new` a un tipo de parámetro cuando una clase genérica cree otras instancias del tipo, tal y como se muestra en el ejemplo siguiente:

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

Cuando use la restricción `new()` con otras restricciones, se debe especificar en último lugar:

```
public class ItemFactory2<T>
    where T : IComparable, new()
{  }
```

Para obtener más información, vea Restricciones de tipos de parámetros.

También puede usar la palabra clave `new` para crear una instancia de un tipo o como un modificador de declaración de miembro.

Especificación del lenguaje C#

Para más información, vea la sección Restricciones de parámetros de tipo de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Genéricos

where (restricción de tipo genérico) (Referencia de C#)

29/06/2020 • 8 minutes to read • Edit Online

La cláusula `where` en una definición genérica especifica restricciones en los tipos que se usan como argumentos para los parámetros de tipo en un tipo genérico, método, delegado o función local. Las restricciones pueden especificar interfaces o clases base, o bien requerir que un tipo genérico sea una referencia, un valor o un tipo no administrado. Declaran las funcionalidades que debe tener el argumento de tipo.

Por ejemplo, se puede declarar una clase genérica, `MyGenericClass`, de modo que el parámetro de tipo `T` implemente la interfaz IComparable<T>:

```
public class AGenericClass<T> where T : IComparable<T> { }
```

> **NOTE**
> Para obtener más información sobre la cláusula where en una expresión de consulta, vea where (Cláusula).

La cláusula `where` también puede incluir una restricción de clase base. La restricción de clase base indica que un tipo que se va a usar como argumento de tipo para ese tipo genérico tiene la clase especificada como clase base, o bien es la clase base. Si se usa la restricción de clase base, debe aparecer antes que cualquier otra restricción de ese parámetro de tipo. Algunos tipos no están permitidos como restricción de clase base: Object, Array y ValueType. Antes de C# 7.3, tampoco se permitían Enum, Delegate ni MulticastDelegate como restricciones de clase base. En el ejemplo siguiente se muestran los tipos que ahora se pueden especificar como clase base:

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

En un contexto que admite un valor NULL en C# 8.0 y versiones posteriores, se aplica la nulabilidad del tipo de clase base. Si la clase base no acepta valores NULL (por ejemplo, `Base`), el argumento de tipo no debe aceptar valores NULL. Si la clase base admite un valor NULL (por ejemplo, `Base?`), el argumento de tipo puede ser un tipo de referencia que acepte o no valores NULL. El compilador emite una advertencia si el argumento de tipo es un tipo de referencia que admite un valor NULL cuando la clase base no acepta valores NULL.

La cláusula `where` puede especificar que el tipo es `class` o `struct`. La restricción `struct` elimina la necesidad de especificar una restricción de clase base de `System.ValueType`. El tipo `System.ValueType` no se puede usar como restricción de clase base. En el ejemplo siguiente se muestran las restricciones `class` y `struct`:

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

En un contexto que admite un valor NULL en C# 8.0 y versiones posteriores, la restricción `class` requiere que un tipo sea un tipo de referencia que no acepte valores NULL. Para permitir tipos de referencia que admitan un valor

NULL, use la restricción `class?`, que permite tipos de referencia que aceptan y que no aceptan valores NULL.

La cláusula `where` puede incluir la restricción `notnull`. La restricción `notnull` limita el parámetro de tipo a tipos que no aceptan valores NULL. Ese tipo puede ser un tipo de valor o un tipo de referencia que no acepta valores NULL. La restricción `notnull` está disponible a partir C# 8.0 para el código compilado en un contexto `nullable enable`. A diferencia de otras restricciones, si un argumento de tipo infringe la restricción `notnull`, el compilador genera una advertencia en lugar de un error. Las advertencias solo se generan en un contexto `nullable enable`.

> **IMPORTANT**
> Las declaraciones genéricas que incluyen la restricción `notnull` se pueden usar en un contexto donde se desconoce que se aceptan valores NULL, pero el compilador no aplica la restricción.

```
#nullable enable
    class NotNullContainer<T>
        where T : notnull
    {
    }
#nullable restore
```

La cláusula `where` también podría incluir una restricción `unmanaged`. La restricción `unmanaged` limita el parámetro de tipo a los tipos conocidos como tipos no administrados. La restricción `unmanaged` hace que sea más fácil escribir código de interoperabilidad de bajo nivel en C#. Esta restricción habilita las rutinas reutilizables en todos los tipos no administrados. La restricción `unmanaged` no se puede combinar con las restricciones `class` o `struct`. La restricción `unmanaged` exige que el tipo sea `struct`:

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

La cláusula `where` también podría incluir una restricción de constructor, `new()`. Esta restricción hace posible crear una instancia de un parámetro de tipo con el operador `new`. La restricción new() permite que el compilador sepa que cualquier argumento de tipo especificado debe tener accesible un constructor sin parámetros. Por ejemplo:

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

La restricción `new()` aparece en último lugar en la cláusula `where`. La restricción `new()` no se puede combinar con las restricciones `struct` o `unmanaged`. Todos los tipos que cumplan esas restricciones deben tener un constructor sin parámetros accesible, lo que hace que la restricción `new()` sea redundante.

Con varios parámetros de tipo, use una cláusula `where` para cada parámetro de tipo, por ejemplo:

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

También puede asociar restricciones a parámetros de tipo de métodos genéricos, como se muestra en el ejemplo siguiente:

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Observe que la sintaxis para describir las restricciones de parámetro de tipo en delegados es igual que la de métodos:

```
delegate T MyDelegate<T>() where T : new();
```

Para obtener información sobre los delegados genéricos, vea Delegados genéricos.

Para obtener más información sobre la sintaxis y el uso de restricciones, vea Restricciones de tipos de parámetros.

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Introducción a los genéricos
- new (restricción)
- Restricciones de tipos de parámetros

base (Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

La palabra clave `base` se usa para acceder a los miembros de la clase base desde una clase derivada:

- Llamar a un método en la clase base que haya sido reemplazado por otro método.
- Especificar a qué constructor de clase base se debe llamar cuando se crean instancias de la clase derivada.

Solo se permite el acceso a una clase base en un constructor, un método de instancia o un descriptor de acceso de propiedad de instancia.

Usar la palabra clave `base` desde dentro de un método estático constituye un error.

La clase base a la que se obtiene acceso es la especificada en la declaración de clase. Por ejemplo, si especifica `class ClassB : ClassA`, se obtiene acceso a los miembros de ClassA desde ClassB, independientemente de la clase base de ClassA.

Ejemplo

En este ejemplo, la clase base, `Person`, y la clase derivada, `Employee`, tienen un método denominado `Getinfo`. Mediante el uso de la palabra clave `base`, es posible llamar al método `Getinfo` en la clase base desde la clase derivada.

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

Para ver más ejemplos, consulte new, virtual y override.

Ejemplo

En este ejemplo se muestra cómo especificar el constructor de clase base al que se llama al crear instancias de una clase derivada.

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- this

this (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `this` hace referencia a la instancia actual de la clase y también se usa como modificador del primer parámetro de un método de extensión.

> **NOTE**
> En este artículo se describe el uso de `this` con instancias de clase. Para obtener más información sobre su uso en métodos de extensión, vea Métodos de extensión.

A continuación se indican usos habituales de `this`:

- Para calificar a miembros ocultos por nombres similares, por ejemplo:

```
public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- Para pasar un objeto como parámetro a otros métodos, por ejemplo:

```
CalcTax(this);
```

- Para declarar indizadores, por ejemplo:

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

Las funciones miembro estáticas no tienen un puntero `this`, debido a que existen en el nivel de clase y no como parte de un objeto. Es un error hacer referencia a `this` en un método estático.

Ejemplo

En este ejemplo, se usa `this` para calificar los miembros de la clase `Employee`, `name` y `alias`, que están ocultos por nombres similares. También se usa para pasar un objeto al método `CalcTax`, que pertenece a otra clase.

```
class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
 */
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#

- Palabras clave de C#
- base
- Métodos

null (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `null` es un literal que representa una referencia nula que no hace referencia a ningún objeto. `null` es el valor predeterminado de las variables de tipo de referencia. Los tipos de valor normales no pueden ser NULL, excepto los tipos de valor que admiten un valor NULL.

En el ejemplo siguiente se muestran algunos comportamientos de la palabra clave `null`:

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals": "does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Palabras clave de C#
- Valores predeterminados de los tipos de C#
- Nothing (Visual Basic)

bool (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave de tipo `bool` es un alias para el tipo de estructura de .NET System.Boolean que representa un valor booleano que puede ser `true` o `false`.

Para realizar operaciones lógicas con valores del tipo `bool`, use operadores lógicos booleanos. El tipo `bool` es el tipo de resultado de los operadores de comparación e igualdad. Una expresión `bool` puede ser una expresión condicional de control en las instrucciones if, do, while y for, así como en el operador condicional `?:`.

El valor predeterminado del tipo `bool` es `false`.

Literales

Puede usar los literales `true` y `false` para inicializar una variable `bool` o para pasar un valor `bool`:

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not checked
```

Lógica booleana de tres valores

Use el tipo `bool?` que acepta valores NULL si tiene que admitir la lógica de tres valores (por ejemplo, si trabaja con bases de datos que admiten un tipo booleano de tres valores). En el caso de los operandos `bool?`, los operadores predefinidos `&` y `|` admiten la lógica de tres valores. Para más información, consulte la sección Operadores lógicos booleanos que aceptan valores NULL del artículo Operadores lógicos booleanos.

Para más información sobre los tipos de valor que admiten un valor NULL, consulte Tipos de valor que admiten un valor NULL.

Conversiones

C# solo proporciona dos conversiones que implican al tipo `bool`. Son una conversión implícita al tipo `bool?` que acepta valores NULL correspondiente y una conversión explícita del tipo `bool?`. Sin embargo, .NET proporciona métodos adicionales que se pueden usar para realizar una conversión al tipo `bool`, o bien revertirla. Para obtener más información, vea la sección Convertir a y desde valores booleanos de la página de referencia de la API System.Boolean.

especificación del lenguaje C#

Para obtener más información, vea la sección Tipo bool de la especificación del lenguaje C#.

Vea también

- Referencia de C#
- Tipos de valor
- Operadores true y false

default (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `default` se puede usar de dos maneras:

- Para especificar la etiqueta predeterminada en la instrucción `switch`.
- Como el operador predeterminado o literal para generar el valor predeterminado de un tipo.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#

Palabras clave contextuales (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Las palabras clave contextuales se usan para proporcionar un significado específico en el código, pero no son una palabra reservada en C#. En esta sección se presentan las siguientes palabras clave contextuales:

PALABRA CLAVE	DESCRIPTION
add	Define un descriptor de acceso de eventos personalizado que se invoca cuando el código de cliente se suscribe al evento.
async	Indica que el método modificado, una expresión lambda o un método anónimo es asincrónico.
await	Suspende un método asincrónico hasta que finalice una tarea en espera.
dynamic	Define un tipo de referencia que permite que las operaciones en las que tiene lugar omitan la comprobación de tipos en tiempo de compilación.
get	Define un método de descriptor de acceso para una propiedad o un indizador.
global	Alias del espacio de nombres global, que de lo contrario no tiene nombre.
partial	Define clases parciales, structs e interfaces a lo largo de la misma unidad de compilación.
remove	Define un descriptor de acceso de eventos personalizado que se invoca cuando el código de cliente cancela la suscripción al evento.
set	Define un método de descriptor de acceso para una propiedad o un indizador.
value	Se usa para establecer descriptores de acceso y para agregar o quitar controladores de eventos.
var	Permite que el compilador determine el tipo de una variable declarada en el ámbito del método.
when	Especifica una condición de filtro para un bloque `catch` o la etiqueta `case` de una instrucción `switch`.
where	Agrega restricciones a una declaración genérica (vea también where).
yield	Se usa en un bloque de iteradores para devolver un valor al objeto enumerador o para señalar el final de la iteración.

Todas las palabras clave de consulta introducidas en C# 3.0 también son contextuales. Para obtener más información, vea Palabras clave para consultas (LINQ).

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#

add (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `add` se usa para definir un descriptor de acceso de eventos personalizado que se invoca cuando el código de cliente se suscribe a su evento. Si proporciona un descriptor de acceso `add` personalizado, también debe proporcionar un descriptor de acceso remove.

Ejemplo

En el ejemplo siguiente se muestra un evento que tiene descriptores de acceso `add` y remove personalizados. Para obtener el ejemplo completo, consulte Procedimiento Implementar eventos de interfaz.

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

Normalmente, no necesita proporcionar sus propios descriptores de acceso de eventos personalizados. Los descriptores de acceso que se generan automáticamente mediante el compilador cuando declara un evento son suficientes para la mayoría de escenarios.

Vea también

- Eventos

get (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `get` define un método de *descriptor de acceso* en una propiedad o un indizador que devuelve el valor de la propiedad o el elemento del indizador. Para obtener más información, consulte Propiedades, Propiedades autoimplementadas e Indexers Indizadores.

En el ejemplo siguiente se definen unos descriptores de acceso `get` y `set` para una propiedad denominada `Seconds`. Usa un campo privado denominado `_seconds` para respaldar el valor de la propiedad.

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

A menudo, el descriptor de acceso `get` consta de una única instrucción que devuelve un valor, como en el ejemplo anterior. A partir de C# 7.0, se puede implementar el descriptor de acceso `get` como un miembro con forma de expresión. En el ejemplo siguiente se implementan los descriptores de acceso `get` y `set` como miembros con forma de expresión.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

En los casos sencillos en los que los descriptores de acceso `get` y `set` de una propiedad no realizan ninguna operación aparte de establecer o recuperar un valor en un campo de respaldo privado, puede aprovechar la compatibilidad del compilador de C# con las propiedades implementadas automáticamente. En el ejemplo siguiente se implementa `Hours` como una propiedad implementada automáticamente.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Propiedades

Tipo parcial (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Las definiciones de tipo parcial permiten dividir la definición de una clase, estructura o interfaz en varios archivos.

En *File1.cs*:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

La declaración en *File2.cs*:

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

Comentarios

Dividir un tipo de clase, estructura o interfaz en varios archivos puede resultar útil cuando trabaja con proyectos de gran tamaño o con código generado automáticamente, como el proporcionado por el Diseñador de Windows Forms. Un tipo parcial puede contener un método parcial. Para obtener más información, consulte Clases y métodos parciales.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Modificadores
- Introducción a los genéricos

partial (Método) (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Un método parcial tiene su signatura definida en una parte de un tipo parcial y su implementación definida en otra parte del tipo. Los métodos parciales permiten a los diseñadores de clases proporcionar enlaces de método, similares a los controladores de eventos, que los desarrolladores pueden decidir implementar o no. Si el desarrollador no proporciona una implementación, el compilador quita la signatura en tiempo de compilación. Se aplican las siguientes condiciones a los métodos parciales:

- Las signaturas de ambas partes del tipo parcial deben coincidir.
- El método debe devolver el valor void.
- No se permiten modificadores de acceso. Los métodos parciales son privados implícitamente.

En el ejemplo siguiente se muestra un método parcial definido en dos partes de una clase parcial:

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

Para obtener más información, consulte Clases y métodos parciales.

Vea también

- Referencia de C#
- partial (tipo)

remove (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `remove` se usa para definir un descriptor de acceso de eventos personalizado que se invoca cuando el código de cliente cancela la suscripción a su evento. Si proporciona un descriptor de acceso `remove` personalizado, también debe proporcionar un descriptor de acceso add.

Ejemplo

En el ejemplo siguiente, se muestra un evento con descriptores de acceso add y `remove` personalizados. Para obtener el ejemplo completo, consulte Procedimiento Implementar eventos de interfaz.

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

Normalmente, no necesita proporcionar sus propios descriptores de acceso de eventos personalizados. Los descriptores de acceso que se generan automáticamente mediante el compilador cuando declara un evento son suficientes para la mayoría de escenarios.

Vea también

- Eventos

set (Referencia de C#)

29/06/2020 • 2 minutes to read • Edit Online

La palabra clave `set` define un método de *descriptor de acceso* en una propiedad o indexador que asigna el valor de la propiedad o del elemento del indexador. Para obtener más información y ejemplos, vea Propiedades, Propiedades autoimplementadas e Indexadores.

En el ejemplo siguiente se definen unos descriptores de acceso `get` y `set` para una propiedad denominada `Seconds`. Usa un campo privado denominado `_seconds` para respaldar el valor de la propiedad.

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

A menudo, el descriptor de acceso `set` consta de una única instrucción que asigna un valor, como en el ejemplo anterior. A partir de C# 7.0, se puede implementar el descriptor de acceso `set` como un miembro con forma de expresión. En el ejemplo siguiente se implementan los descriptores de acceso `get` y `set` como miembros con forma de expresión.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

En los casos sencillos en los que los descriptores de acceso `get` y `set` de una propiedad no realizan ninguna operación aparte de establecer o recuperar un valor en un campo de respaldo privado, puede aprovechar la compatibilidad del compilador de C# con las propiedades implementadas automáticamente. En el ejemplo siguiente se implementa `Hours` como una propiedad implementada automáticamente.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#
- Propiedades

when (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

Puede usar la palabra clave contextual `when` para especificar una condición de filtro en dos contextos:

- En la instrucción `catch` de un bloque try/catch o try/catch/finally.
- En la etiqueta `case` de una instrucción switch.

`when` en una instrucción `catch`

A partir de C# 6, puede usarse `when` en una instrucción `catch` para especificar una condición que debe cumplirse para que el controlador de una excepción específica se ejecute. Su sintaxis es:

```
catch (ExceptionType [e]) when (expr)
```

donde *expr* es una expresión que se evalúa como un valor booleano. Si devuelve `true`, el controlador de excepciones se ejecuta; si devuelve `false`, no se ejecuta.

En el ejemplo siguiente se usa la palabra clave `when` para ejecutar condicionalmente controladores para una HttpRequestException según el texto del mensaje de excepción.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
        var streamTask = client.GetStringAsync("https://localHost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}
```

`when` en una instrucción `switch`

A partir de C# 7.0, las etiquetas `case` ya no tienen que ser mutuamente exclusivas y el orden con el que las etiquetas `case` aparecen en una instrucción `switch` puede determinar el bloque switch que se ejecuta. Puede usarse la palabra clave `when` para especificar una condición de filtro que haga que su etiqueta case asociada se cumpla únicamente si también se cumple la condición de filtro. Su sintaxis es:

```
case (expr) when (when-condition):
```

donde *expr* es un patrón de constante o un patrón de tipo que se compara con la expresión match y *when-condition* es cualquier expresión booleana.

En el ejemplo siguiente se usa la palabra clave `when` para probar objetos `Shape` que tienen un área de cero, así como para probar una serie de objetos `Shape` que tienen un área mayor que cero.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }
   public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public double Length { get; set; }
   public double Width { get; set; }

   public override double Area
   {
      get { return Math.Round(Length * Width,2); }
   }

   public override double Circumference
   {
      get { return (Length + Width) * 2; }
   }
}

public class Square : Rectangle
{
   public Square(double side) : base(side, side)
   {
      Side = side;
   }

   public double Side { get; set; }
}

public class Example
{
   public static void Main()
   {
      Shape sh = null;
      Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                         new Rectangle(10, 10), sh, new Square(0) };
      foreach (var shape in shapes)
```

```
        ShowShapeInfo(shape);
    }

    private static void ShowShapeInfo(Object obj)
    {
       switch (obj)
       {
          case Shape shape when shape.Area == 0:
             Console.WriteLine($"The shape: {shape.GetType().Name} with no dimensions");
             break;
          case Square sq when sq.Area > 0:
             Console.WriteLine("Information about the square:");
             Console.WriteLine($"   Length of a side: {sq.Side}");
             Console.WriteLine($"   Area: {sq.Area}");
             break;
          case Rectangle r when r.Area > 0:
             Console.WriteLine("Information about the rectangle:");
             Console.WriteLine($"   Dimensions: {r.Length} x {r.Width}");
             Console.WriteLine($"   Area: {r.Area}");
             break;
          case Shape shape:
             Console.WriteLine($"A {shape.GetType().Name} shape");
             break;
          case null:
             Console.WriteLine($"The {nameof(obj)} variable is uninitialized.");
             break;
          default:
             Console.WriteLine($"The {nameof(obj)} variable does not represent a Shape.");
             break;
       }
    }
}
// The example displays the following output:
//       Information about the square:
//          Length of a side: 10
//          Area: 100
//       Information about the rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//       Information about the rectangle:
//          Dimensions: 10 x 10
//          Area: 100
//       The obj variable is uninitialized.
//       The shape: Square with no dimensions
```

Vea también

- switch (Instrucción)
- try/catch (Instrucción)
- try/catch/finally (Instrucción)

value (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `value` se usa en el descriptor de acceso `set` de las declaraciones propiedad y indizador. Es parecido a un parámetro de entrada de un método. El término `value` hace referencia al valor que el código de cliente intenta asignar a la propiedad o indizador. En el ejemplo siguiente, `MyDerivedClass` tiene una propiedad denominada `Name` que usa el parámetro `value` para asignar una nueva cadena al campo de respaldo `name`. Desde el punto de vista del código de cliente, la operación se escribe como una simple asignación.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
```

Para obtener más información, consulte los artículos Propiedades e Indizadores.

especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- Palabras clave de C#

yield (Referencia de C#)

29/06/2020 • 7 minutes to read • Edit Online

Cuando se usa la `yield` palabra clave contextual en una instrucción, se indica que el método, el operador o el descriptor de acceso `get` en el que aparece es un iterador. Al usar `yield` para definir un iterador ya no es necesaria una clase adicional explícita (la clase que retiene el estado para una enumeración, consulte IEnumerator<T> para ver un ejemplo) al implementar los patrones IEnumerable y IEnumerator para un tipo de colección personalizado.

En el ejemplo siguiente se muestran las dos formas de la instrucción `yield`.

```
yield return <expression>;
yield break;
```

Comentarios

Utilice una instrucción `yield return` para devolver cada elemento de uno en uno.

La secuencia devuelta desde un método iterador se puede consumir mediante la instrucción foreach o una consulta LINQ. Cada iteración del bucle `foreach` llama al método iterador. Cuando se alcanza una instrucción `yield return` en el método iterador, se devuelve `expression` y se conserva la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama a la función del iterador.

Puede usar una instrucción `yield break` para finalizar la iteración.

Para obtener más información sobre los iteradores, vea Iteradores.

Métodos de iterador y descriptores de acceso get

La declaración de un iterador debe cumplir los requisitos siguientes:

- El tipo de valor devuelto debe ser IEnumerable, IEnumerable<T>, IEnumerator o IEnumerator<T>.
- La declaración no puede tener ningún parámetro in ref o out.

El tipo `yield` de un iterador que devuelve IEnumerable o IEnumerator es `object`. Si el iterador devuelve IEnumerable<T> o IEnumerator<T>, debe haber una conversión implícita del tipo de la expresión en la instrucción `yield return` al parámetro de tipo genérico.

No puede incluir una instrucción `yield return` ni `yield break` en:

- Expresiones lambda y métodos anónimos.
- Métodos que contienen bloques inseguros. Para obtener más información, vea unsafe.

Control de excepciones

Una instrucción `yield return` no puede encontrarse en un bloque try-catch. Una instrucción `yield return` puede encontrarse en el bloque try de una instrucción try-finally.

Una instrucción `yield break` puede encontrarse en un bloque try o un bloque catch, pero no en un bloque finally.

Si el cuerpo `foreach` (fuera del método iterador) produce una excepción, se ejecuta un bloque `finally` en el método iterador.

Implementación técnica

El código siguiente devuelve un valor `IEnumerable<string>` desde un método iterador y, a continuación, recorre sus elementos en iteración.

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
   ...
}
```

La llamada a `MyIteratorMethod` no ejecuta el cuerpo del método. En su lugar, la llamada devuelve un valor `IEnumerable<string>` en la variable `elements`.

En una iteración del bucle `foreach`, se llama al método MoveNext para `elements`. Esta llamada ejecuta el cuerpo de `MyIteratorMethod` hasta que se alcanza la siguiente instrucción `yield return`. La expresión devuelta por la instrucción `yield return` determina no solo el valor de la variable `element` para que la utilice el cuerpo del bucle, sino también la propiedad Current de `elements`, que es un valor `IEnumerable<string>`.

En cada iteración subsiguiente del bucle `foreach`, la ejecución del cuerpo del iterador continúa desde donde se dejó, deteniéndose de nuevo al alcanzar una instrucción `yield return`. El bucle `foreach` se completa al alcanzar el fin del método iterador o una instrucción `yield break`.

Ejemplo

El ejemplo siguiente tiene una instrucción `yield return` que está dentro de un bucle `for`. Cada iteración del cuerpo de instrucción `foreach` en el método `Main` crea una llamada a la función de iterador `Power`. Cada llamada a la función de iterador prosigue con la siguiente ejecución de la instrucción `yield return`, que se produce durante la siguiente iteración del bucle `for`.

El tipo de valor devuelto del método iterador es IEnumerable, que es un tipo de interfaz de iteradores. Cuando se llama al método iterador, este devuelve un objeto enumerable que contiene las potencias de un número.

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

Ejemplo

En el ejemplo siguiente se muestra un descriptor de acceso `get` que es un iterador. En el ejemplo, cada una de las instrucciones `yield return` devuelve una instancia de una clase definida por el usuario.

```
public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}
```

Especificación del lenguaje C#

Para obtener más información, consulte la Especificación del lenguaje C#. La especificación del lenguaje es la fuente definitiva de la sintaxis y el uso de C#.

Vea también

- Referencia de C#
- Guía de programación de C#
- foreach, in
- Iteradores

Palabras clave de consulta (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

En esta sección, se incluyen las palabras clave contextuales que se usan en expresiones de consulta.

En esta sección

CLÁUSULA	DESCRIPTION
from	Especifica un origen de datos y una variable de rango (similar a una variable de iteración).
where	Filtra los elementos de origen en función de una o varias expresiones booleanas separadas por operadores lógicos AND y OR (`&&` o `||`).
select	Especifica el tipo y la forma que tendrán los elementos de la secuencia devuelta cuando se ejecute la consulta.
group	Agrupa los resultados de la consulta según un valor clave especificado.
into	Proporciona un identificador que puede actuar como una referencia a los resultados de una combinación, un grupo o una cláusula select.
orderby	Ordena los resultados de las consultas en orden ascendente o descendente según el comparador predeterminado del tipo de elemento.
join	Combina dos orígenes de datos en función de una comparación de igualdad entre dos criterios de coincidencia especificados.
let	Presenta una variable de rango para almacenar los resultados de la subexpresión en una expresión de consulta.
in	Palabra clave contextual en una cláusula join.
on	Palabra clave contextual en una cláusula join.
equals	Palabra clave contextual en una cláusula join.
by	Palabra clave contextual en una cláusula group.
ascending	Palabra clave contextual en una cláusula orderby.
descending	Palabra clave contextual en una cláusula orderby.

Vea también

- Palabras clave de C#
- LINQ (Language Integrated Query)
- LINQ en C#

from (Cláusula, Referencia de C#)

18/03/2020 • 8 minutes to read • Edit Online

Una expresión de consulta debe comenzar con una cláusula `from`, Además, una expresión de consulta puede contener subconsultas, que también comienzan con una cláusula `from`. La cláusula `from` especifica lo siguiente:

- El origen de datos en el que se ejecutará la consulta o subconsulta.
- Una *variable de rango* local que representa cada elemento de la secuencia de origen.

Tanto la variable de rango como el origen de datos están fuertemente tipados. El origen de datos al que se hace referencia en la cláusula `from` debe tener un tipo de IEnumerable, IEnumerable<T> o un tipo derivado como IQueryable<T>.

En el ejemplo siguiente, `numbers` es el origen de datos y `num` es la variable de rango. Tenga en cuenta que ambas variables están fuertemente tipadas a pesar de que se usa la palabra clave var.

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

Variable de rango

El compilador deduce el tipo de la variable de rango cuando el origen de datos implementa IEnumerable<T>. Por ejemplo, si el origen tiene un tipo de `IEnumerable<Customer>`, entonces se deduce que la variable de rango es `Customer`. La única vez en que debe especificar el tipo explícitamente es cuando el origen es un tipo `IEnumerable` no genérico como ArrayList. Para obtener más información, vea Procedimiento para consultar un objeto ArrayList con LINQ (C#).

En el ejemplo anterior, `num` se deduce que es de tipo `int`. Como la variable de rango está fuertemente tipada, puede llamar a los métodos en ella o usarla en otras operaciones. Por ejemplo, en lugar de escribir `select num`, podría escribir `select num.ToString()` para hacer que la expresión de consulta devuelva una secuencia de cadenas en lugar de enteros. O podría escribir `select num + 10` para hacer que la expresión devuelva la secuencia 14, 11, 13, 12, 10. Para obtener más información, vea Cláusula select.

La variable de rango es como una variable de iteración en una instrucción foreach excepto por una diferencia

muy importante: una variable de rango realmente nunca almacena datos del origen. Es solo una comodidad sintáctica que permite a la consulta describir lo que ocurrirá cuando se ejecute la consulta. Para obtener más información, vea Introduction to LINQ queries (C#) [Introducción a las consultas LINQ (C#)].

Cláusulas From compuestas

En algunos casos, cada elemento de la secuencia de origen puede ser por sí mismo una secuencia o contener una. Por ejemplo, su origen de datos puede ser un `IEnumerable<Student>` donde cada objeto de estudiante en la secuencia contiene una lista de resultados de las pruebas. Para tener acceso a la lista interna dentro de cada elemento `Student`, puede usar las cláusulas `from` compuestas. La técnica es como usar instrucciones foreach anidadas. Puede agregar cláusulas where u orderby a cualquier cláusula `from` para filtrar los resultados. En el ejemplo siguiente se muestra una secuencia de objetos `Student`, cada uno de los cuales contiene un `List` interno de enteros que representa los resultados de las pruebas. Para tener acceso a la lista interna, use una cláusula `from` compuesta. Puede insertar cláusulas entre las dos cláusulas `from` si es necesario.

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},
           new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},
           new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},
           new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
           new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // Use a compound from to access the inner sequence within each element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                         from score in student.Scores
                            where score > 90
                            select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/
```

Usar cláusulas From múltiples para realizar combinaciones

Una cláusula `from` compuesta se usa para tener acceso a las colecciones internas en un origen de datos único. En cambio, una consulta también puede contener varias cláusulas `from` que generan consultas adicionales de orígenes de datos independientes. Esta técnica le permite realizar determinados tipos de operaciones de combinación que no son posibles mediante la cláusula join.

En el siguiente ejemplo se muestra cómo pueden usarse dos cláusulas `from` para formar una combinación

cruzada completa de dos orígenes de datos.

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
        Cross join:
        A is matched to x
        A is matched to y
        A is matched to z
        B is matched to x
        B is matched to y
        B is matched to z
        C is matched to x
        C is matched to y
        C is matched to z
        Filtered non-equijoin:
        y is matched to A
        y is matched to B
        y is matched to C
        z is matched to A
        z is matched to B
        z is matched to C
        */
```

Para obtener más información sobre las operaciones de combinación que usan varias cláusulas `from`, vea Realizar operaciones de combinación externa izquierda.

Vea también

- Palabras clave para consultas (LINQ)
- Language-Integrated Query (LINQ)

where (Cláusula, Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

La cláusula `where` se usa en una expresión de consulta para especificar los elementos del origen de datos que se devuelven en dicha expresión. Aplica una condición booleana (*predicate*) a cada elemento de origen (al que hace referencia la variable de rango) y devuelve aquellos en los que la condición especificada se cumple. Puede que una sola expresión de consulta contenga varias cláusulas `where` y que una sola cláusula contenga varias subexpresiones de predicado.

Ejemplo

En el ejemplo siguiente, la cláusula `where` filtra todos los números excepto los que son inferiores a cinco. Si la cláusula `where` se quita, se devolverán todos los números del origen de datos. La expresión `num < 5` es el predicado que se aplica a cada elemento.

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

Ejemplo

En una sola cláusula `where`, se pueden especificar todos los predicados que sean necesarios mediante los operadores `&&` y `||`. En el ejemplo siguiente, la consulta especifica dos predicados para seleccionar únicamente los números pares que sean inferiores a cinco.

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
// Output:
// 4 2 0
// 4 2 0
```

Ejemplo

Puede que una cláusula `where` contenga uno o más métodos que devuelvan valores booleanos. En el ejemplo siguiente, la cláusula `where` usa un método para determinar si el valor actual de la variable de rango es par o impar.

```
class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0
```

Comentarios

La cláusula `where` es un mecanismo de filtrado. Se puede colocar prácticamente en cualquier parte en una expresión de consulta, pero no puede ser la primera ni la última cláusula. Puede que una cláusula `where` aparezca antes o después de una cláusula group, en función de que haya que filtrar los elementos de origen antes o después de agruparlos.

Si un predicado especificado no es válido para los elementos del origen de datos, se producirá un error en tiempo de compilación. Esta es una de las ventajas de la comprobación fuertemente tipada que ofrece LINQ.

En tiempo de compilación, la palabra clave `where` se convierte en una llamada al método de operador de consulta estándar Where.

Vea también

- Palabras clave para consultas (LINQ)
- from (cláusula)
- select (cláusula)
- Filtrado de datos
- LINQ en C#
- Language-Integrated Query (LINQ)

select (Cláusula, Referencia de C#)

18/03/2020 • 7 minutes to read • Edit Online

En una expresión de consulta, la cláusula `select` especifica el tipo de valores que se producirán cuando se ejecute la consulta. El resultado se basa en la evaluación de todas las cláusulas anteriores y en cualquier expresión de la propia cláusula `select`. Una expresión de consulta debe finalizar con una cláusula `select` o con una cláusula group.

En el ejemplo siguiente, se muestra una cláusula `select` simple en una expresión de consulta.

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

El tipo de la secuencia generada por la cláusula `select` determina el tipo de la variable de consulta `queryHighScores`. En el caso más simple, la cláusula `select` simplemente especifica la variable de rango. Esto hace que la secuencia devuelta contenga elementos del mismo tipo que el origen de datos. Para obtener más información, vea Relaciones entre tipos en operaciones de consulta LINQ. En cambio, la cláusula `select` también proporciona un mecanismo eficaz para transformar (o *proyectar*) el origen de datos en nuevos tipos. Para obtener más información, vea Transformaciones de datos con LINQ (C#).

Ejemplo

En el ejemplo siguiente, se muestran las distintas formas que puede tener una cláusula `select`. En cada consulta, tenga en cuenta la relación entre la cláusula `select` y el tipo de la *variable de consulta* (`studentQuery1`, `studentQuery2`, etc.).

```
    class SelectSample2
    {
        // Define some classes
        public class Student
        {
            public string First { get; set; }
            public string Last { get; set; }
            public int ID { get; set; }
            public List<int> Scores;
            public ContactInfo GetContactInfo(SelectSample2 app, int id)
            {
```

```
            {
                ContactInfo cInfo =
                    (from ci in app.contactList
                    where ci.ID == id
                    select ci)
                    .FirstOrDefault();

                return cInfo;
            }

            public override string ToString()
            {
                return First + " " + Last + ":" + ID;
            }
        }

        public class ContactInfo
        {
            public int ID { get; set; }
            public string Email { get; set; }
            public string Phone { get; set; }
            public override string ToString() { return Email + "," + Phone; }
        }

        public class ScoreInfo
        {
            public double Average { get; set; }
            public int ID { get; set; }
        }

        // The primary data source
        List<Student> students = new List<Student>()
        {
             new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92,
81, 60}},
             new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91,
39}},
             new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65,
91}},
             new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85,
82}},
        };

        // Separate data source for contact info.
        List<ContactInfo> contactList = new List<ContactInfo>()
        {
            new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-555-0108"},
            new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-555-0298"},
            new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-555-1130"},
            new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-555-0521"}
        };

        static void Main(string[] args)
        {
            SelectSample2 app = new SelectSample2();

            // Produce a filtered sequence of unmodified Students.
            IEnumerable<Student> studentQuery1 =
                from student in app.students
                where student.ID > 111
                select student;

            Console.WriteLine("Query1: select range_variable");
            foreach (Student s in studentQuery1)
            {
                Console.WriteLine(s.ToString());
            }

            // Produce a filtered sequence of elements that contain
            // only one property of each Student
```

```
// Only one property of each Student.
IEnumerable<String> studentQuery2 =
    from student in app.students
    where student.ID > 111
    select student.Last;

Console.WriteLine("\r\n studentQuery2: select range_variable.Property");
foreach (string s in studentQuery2)
{
    Console.WriteLine(s);
}

// Produce a filtered sequence of objects created by
// a method call on each Student.
IEnumerable<ContactInfo> studentQuery3 =
    from student in app.students
    where student.ID > 111
    select student.GetContactInfo(app, student.ID);

Console.WriteLine("\r\n studentQuery3: select range_variable.Method");
foreach (ContactInfo ci in studentQuery3)
{
    Console.WriteLine(ci.ToString());
}

// Produce a filtered sequence of ints from
// the internal array inside each Student.
IEnumerable<int> studentQuery4 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0];

Console.WriteLine("\r\n studentQuery4: select range_variable[index]");
foreach (int i in studentQuery4)
{
    Console.WriteLine("First score = {0}", i);
}

// Produce a filtered sequence of doubles
// that are the result of an expression.
IEnumerable<double> studentQuery5 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0] * 1.1;

Console.WriteLine("\r\n studentQuery5: select expression");
foreach (double d in studentQuery5)
{
    Console.WriteLine("Adjusted first score = {0}", d);
}

// Produce a filtered sequence of doubles that are
// the result of a method call.
IEnumerable<double> studentQuery6 =
    from student in app.students
    where student.ID > 111
    select student.Scores.Average();

Console.WriteLine("\r\n studentQuery6: select expression2");
foreach (double d in studentQuery6)
{
    Console.WriteLine("Average = {0}", d);
}

// Produce a filtered sequence of anonymous types
// that contain only two properties from each Student.
var studentQuery7 =
    from student in app.students
    where student.ID > 111
    select new { student.First, student.Last };
```

```
            select new { student.First, student.Last };

        Console.WriteLine("\r\n studentQuery7: select new anonymous type");
        foreach (var item in studentQuery7)
        {
            Console.WriteLine("{0}, {1}", item.Last, item.First);
        }

        // Produce a filtered sequence of named objects that contain
        // a method return value and a property from each Student.
        // Use named types if you need to pass the query variable
        // across a method boundary.
        IEnumerable<ScoreInfo> studentQuery8 =
            from student in app.students
            where student.ID > 111
            select new ScoreInfo
            {
                Average = student.Scores.Average(),
                ID = student.ID
            };

        Console.WriteLine("\r\n studentQuery8: select new named type");
        foreach (ScoreInfo si in studentQuery8)
        {
            Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
        }

        // Produce a filtered sequence of students who appear on a contact list
        // and whose average is greater than 85.
        IEnumerable<ContactInfo> studentQuery9 =
            from student in app.students
            where student.Scores.Average() > 85
            join ci in app.contactList on student.ID equals ci.ID
            select ci;

        Console.WriteLine("\r\n studentQuery9: select result of join clause");
        foreach (ContactInfo ci in studentQuery9)
        {
            Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
        }
    }
/* Output
    Query1: select range_variable
    Claire O'Donnell:112
    Sven Mortensen:113
    Cesar Garcia:114

    studentQuery2: select range_variable.Property
    O'Donnell
    Mortensen
    Garcia

    studentQuery3: select range_variable.Method
    ClaireO@Contoso.com,206-555-0298
    SvenMort@Contoso.com,206-555-1130
    CesarGar@Contoso.com,206-555-0521

    studentQuery4: select range_variable[index]
    First score = 75
    First score = 88
    First score = 97

    studentQuery5: select expression
    Adjusted first score = 82.5
```

```
        Adjusted first score = 96.8
        Adjusted first score = 106.7

        studentQuery6: select expression2
        Average = 72.25
        Average = 84.5
        Average = 88.25

        studentQuery7: select new anonymous type
        O'Donnell, Claire
        Mortensen, Sven
        Garcia, Cesar

        studentQuery8: select new named type
        ID = 112, Average = 72.25
        ID = 113, Average = 84.5
        ID = 114, Average = 88.25

        studentQuery9: select result of join clause
        ID = 114, Email = CesarGar@Contoso.com
*/
```

Como se muestra en `studentQuery8` en el ejemplo anterior, a veces es posible que quiera que los elementos de la secuencia devuelta contengan solo un subconjunto de las propiedades de los elementos de origen. Al mantener la secuencia devuelta lo más pequeña posible, puede reducir los requisitos de memoria y aumentar la velocidad de ejecución de la consulta. Puede hacerlo al crear un tipo anónimo en la cláusula `select` y usar un inicializador de objeto para inicializarlo con las propiedades adecuadas del elemento de origen. Para obtener un ejemplo de cómo hacerlo, consulte Inicializadores de objeto y de colección.

Comentarios

En tiempo de compilación, la cláusula `select` se convierte en una llamada de método al operador de consulta estándar Select.

Vea también

- Referencia de C#
- Palabras clave para consultas (LINQ)
- from (cláusula)
- partial (Método) (Referencia de C#)
- Tipos anónimos
- LINQ en C#
- Language-Integrated Query (LINQ)

group (Cláusula, Referencia de C#)

18/03/2020 • 12 minutes to read • Edit Online

La cláusula `group` devuelve una secuencia de objetos IGrouping<TKey,TElement> que contienen cero o más elementos que coinciden con el valor de clave del grupo. Por ejemplo, puede agrupar una secuencia de cadenas según la primera letra de cada cadena. En este caso, la primera letra es la clave, es de tipo char y se almacena en la propiedad `Key` de cada objeto IGrouping<TKey,TElement>. El compilador deduce el tipo de la clave.

Puede finalizar una expresión de consulta con una cláusula `group`, como se muestra en el ejemplo siguiente:

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

Si quiere realizar operaciones de consulta adicionales en cada grupo, puede especificar un identificador temporal mediante la palabra clave contextual into. Cuando se usa `into`, es necesario continuar con la consulta y finalmente terminarla con una instrucción `select` u otra cláusula `group`, como se muestra en el extracto siguiente:

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

En la sección Ejemplo de este artículo se proporcionan ejemplos más completos sobre el uso de `group` con y sin `into`.

Enumeración de los resultados de una consulta de grupo

Dado que los objetos IGrouping<TKey,TElement> generados por una consulta `group` son esencialmente una lista de listas, debe usar un bucle foreach anidado para tener acceso a los elementos de cada grupo. El bucle exterior recorre en iteración las claves de grupo y el bucle interno recorre en iteración cada elemento del propio grupo. Un grupo puede tener una clave pero ningún elemento. A continuación se muestra el bucle `foreach` que ejecuta la consulta en los ejemplos de código anteriores:

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.First);
     }
 }
```

Tipos de clave

Las claves de grupo pueden ser de cualquier tipo, como una cadena, un tipo numérico integrado, un tipo con nombre definido por el usuario o un tipo anónimo.

Agrupar por cadena

En los ejemplos de código anteriores se ha usado el tipo `char`. En su lugar, podría haberse especificado una clave de cadena, por ejemplo, los apellidos completos:

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

Agrupar por valor booleano

En el ejemplo siguiente se muestra el uso de un valor booleano para una clave con el fin de dividir los resultados en dos grupos. Observe que el valor se genera a partir de una subexpresión en la cláusula `group`.

```
class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91,
39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91,
95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last, student.First,
student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
  Low averages
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Garcia, Cesar:75.5
  High averages
   Mortensen, Sven:93.5
   Garcia, Debra:88.25
*/
```

Agrupar por intervalo numérico

En el ejemplo siguiente se usa una expresión para crear claves de grupo numéricas que representan un intervalo de percentil. Observe el uso de let como ubicación adecuada para almacenar el resultado de una llamada de método, para no tener que llamar al método dos veces en la cláusula `group`. Para más información sobre cómo usar métodos en expresiones de consulta de manera segura, consulte Controlar excepciones en expresiones de consulta.

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91,
39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91,
95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last, student.First,
student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
```

```
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
     Students with an average between 70 and 80
       Omelchenko, Svetlana:77.5
       O'Donnell, Claire:72.25
       Garcia, Cesar:75.5
     Students with an average between 80 and 90
       Garcia, Debra:88.25
     Students with an average between 90 and 100
       Mortensen, Sven:93.5
 */
```

Agrupación por claves compuestas

Use una clave compuesta cuando quiera agrupar los elementos según más de una clave. Para crear una clave compuesta se usa un tipo anónimo o un tipo con nombre para contener el elemento de clave. En el ejemplo siguiente, supongamos que una clase `Person` se ha declarado con los miembros denominados `surname` y `city`. La cláusula `group` hace que se cree un grupo independiente para cada conjunto de personas con el mismo apellido y la misma ciudad.

```
group person by new {name = person.surname, city = person.city};
```

Use un tipo con nombre si debe pasar la variable de consulta a otro método. Cree una clase especial usando las propiedades autoimplementadas para las claves y, luego, invalide los métodos Equals y GetHashCode. También puede usar un struct, en cuyo caso no es estrictamente necesario invalidar esos métodos. Para más información, consulte Procedimiento Implementar una clase ligera con propiedades autoimplementadas y Procedimiento para buscar archivos duplicados en un árbol de directorios. El último artículo contiene un ejemplo de código en el que se muestra cómo usar una clave compuesta con un tipo con nombre.

Ejemplo

En el ejemplo siguiente se muestra el patrón estándar para ordenar los datos de origen en grupos cuando no se aplica ninguna lógica de consulta adicional a los grupos. Esto se denomina agrupación sin continuación. Los elementos de una matriz de cadenas se agrupan por la primera letra. El resultado de la consulta es un tipo IGrouping<TKey,TElement> que contiene una propiedad `Key` pública de tipo `char` y una colección IEnumerable<T> que contiene cada elemento de la agrupación.

El resultado de una cláusula `group` es una secuencia de secuencias. Por consiguiente, para tener acceso a los elementos individuales de cada grupo devuelto, use un bucle `foreach` anidado dentro del bucle que recorre en iteración las claves de grupo, como se muestra en el ejemplo siguiente.

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
      Words that start with the letter 'b':
        blueberry
        banana
      Words that start with the letter 'c':
        chimpanzee
        cheese
      Words that start with the letter 'a':
        abacus
        apple
     */
```

Ejemplo

En este ejemplo se muestra cómo aplicar lógica adicional a los grupos después de haberlos creado, mediante el uso de una *continuación* con `into`. Para obtener más información, vea into. En el ejemplo siguiente se consulta cada grupo para seleccionar solo aquellos cuyo valor de clave sea una vocal.

```
class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese",
"elephant", "umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                   || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("   {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/
```

Comentarios

En tiempo de compilación, las cláusulas `group` se convierten en llamadas al método GroupBy.

Vea también

- IGrouping<TKey,TElement>
- GroupBy
- ThenBy
- ThenByDescending
- Palabras clave para consultas
- Language-Integrated Query (LINQ)
- Crear grupos anidados
- Agrupar los resultados de consultas
- Realizar una subconsulta en una operación de agrupación

into (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `into` puede usarse para crear un identificador temporal para almacenar los resultados de una cláusula group, join o select en un nuevo identificador. Este identificador puede ser un generador de comandos de consulta adicionales. Cuando se usa en una cláusula `group` o `select`, el uso del nuevo identificador se denomina a veces una *continuación*.

Ejemplo

En el ejemplo siguiente, se muestra el uso de la palabra clave `into` para habilitar un identificador temporal `fruitGroup` que tiene un tipo deducido de `IGrouping`. Mediante el identificador, puede invocar el método Count en cada grupo y seleccionar solo los grupos que contienen dos o más palabras.

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots"};

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstLetter, item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

El uso de `into` en una cláusula `group` solo es necesario cuando quiere realizar operaciones de consulta adicionales en cada grupo. Para obtener más información, vea group (Cláusula).

Para obtener un ejemplo del uso de `into` en una cláusula `join`, vea join (Cláusula).

Vea también

- Palabras clave para consultas (LINQ)
- LINQ en C#
- group (cláusula)

orderby (Cláusula, Referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

En una expresión de consulta, la cláusula `orderby` hace que la secuencia o subsecuencia (grupo) devuelta se ordene de forma ascendente o descendente. Se pueden especificar varias claves para llevar a cabo una o varias operaciones de ordenación secundaria. La ordenación se realiza mediante el comparador predeterminado del tipo de elemento. El criterio de ordenación predeterminado es el ascendente. También puede especificar un comparador personalizado. En cambio, solo está disponible mediante la sintaxis basada en métodos. Para obtener más información, consulte Sorting Data (Ordenación de datos).

Ejemplo

En el ejemplo siguiente, la primera consulta ordena las palabras en orden alfabético a partir de la A y la segunda consulta ordena las mismas palabras en orden descendente. (La palabra clave `ascending` es el valor de ordenación predeterminado y puede omitirse).

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

Ejemplo

En el ejemplo siguiente, se realiza una ordenación primaria por apellidos de los alumnos y, después, una ordenación secundaria por sus nombres.

```
class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }
```

```
    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111},
           new Student {First="Claire", Last="O'Donnell", ID=112},
           new Student {First="Sven", Last="Mortensen", ID=113},
           new Student {First="Cesar", Last="Garcia", ID=114},
           new Student {First="Debra", Last="Garcia", ID=115}
        };

        return students;
    }
    static void Main(string[] args)
    {
        // Create the data source.
        List<Student> students = GetStudents();

        // Create the query.
        IEnumerable<Student> sortedStudents =
            from student in students
            orderby student.Last ascending, student.First ascending
            select student;

        // Execute the query.
        Console.WriteLine("sortedStudents:");
        foreach (Student student in sortedStudents)
            Console.WriteLine(student.Last + " " + student.First);

        // Now create groups and sort the groups. The query first sorts the names
        // of all students so that they will be in alphabetical order after they are
        // grouped. The second orderby sorts the group keys in alpha order.
        var sortedGroups =
            from student in students
            orderby student.Last, student.First
            group student by student.Last[0] into newGroup
            orderby newGroup.Key
            select newGroup;

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "sortedGroups:");
        foreach (var studentGroup in sortedGroups)
        {
            Console.WriteLine(studentGroup.Key);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}", student.Last, student.First);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
   Garcia, Cesar
   Garcia, Debra
```

```
   Garcia, Debra
M
   Mortensen, Sven
O
   O'Donnell, Claire
   Omelchenko, Svetlana
*/
```

Comentarios

En tiempo de compilación, la cláusula `orderby` se convierte en una llamada al método OrderBy. Varias claves en la cláusula `orderby` se convierten en llamadas al método ThenBy.

Vea también

- Referencia de C#
- Palabras clave para consultas (LINQ)
- LINQ en C#
- group (cláusula)
- Language-Integrated Query (LINQ)

join (Cláusula, Referencia de C#)

18/03/2020 • 15 minutes to read • Edit Online

La cláusula `join` es útil para asociar elementos de secuencias de origen diferentes que no tienen ninguna relación directa en el modelo de objetos. El único requisito es que los elementos de cada origen compartan algún valor del que se pueda comparar la igualdad. Por ejemplo, imagínese que un distribuidor de comida tiene una lista de proveedores de un determinado producto y una lista de compradores. Se puede usar una cláusula `join`, por ejemplo, para crear una lista de los proveedores y compradores de dicho producto que se encuentran en la misma región especificada.

La cláusula `join` toma dos secuencias de origen como entrada. Los elementos de cada secuencia deben ser o deben contener una propiedad que se pueda comparar con una propiedad correspondiente en la otra secuencia. La cláusula `join` compara la igualdad de las claves especificadas mediante la palabra clave especial `equals`. Todas las combinaciones efectuadas por la cláusula `join` son combinaciones de igualdad. La forma de la salida de una cláusula `join` depende del tipo específico de combinación que se va a efectuar. Estos son los tres tipos de combinación más comunes:

- Combinación interna
- Combinación agrupada
- Combinación externa izquierda

Combinación interna

En el ejemplo siguiente se muestra una combinación de igualdad interna simple. Esta consulta genera una secuencia plana de pares de "nombre de producto y categoría". La misma cadena de categoría aparecerá en varios elementos. Si un elemento de `categories` no tiene ningún `products` que coincida, dicha categoría no aparecerá en los resultados.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

Para obtener más información, vea Realizar combinaciones internas.

Combinación agrupada

Una cláusula `join` con una expresión `into` se denomina "combinación agrupada".

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

Las combinaciones agrupadas generan una secuencia de resultados jerárquicos que asocia los elementos de la secuencia de origen izquierda con uno o más elementos coincidentes de la secuencia de origen derecha. Las combinaciones agrupadas no tienen ningún equivalente en términos relacionales; son básicamente una secuencia de matrices de objetos.

Si no se encuentra ningún elemento de la secuencia de origen derecha para que coincida con un elemento del origen izquierdo, la cláusula `join` generará una matriz vacía para ese elemento. Por lo tanto, la combinación agrupada es básicamente una combinación de igualdad interna, salvo que la secuencia del resultado se organiza en grupos.

Si solo selecciona los resultados de una combinación agrupada, puede tener acceso a los elementos, pero no podrá identificar la clave en la que coinciden. Por lo tanto, suele resultar más útil seleccionar los resultados de la combinación agrupada en un nuevo tipo que también tenga el nombre de la clave, como se muestra en el ejemplo anterior.

Por supuesto, también puede usar el resultado de una combinación agrupada como generador de otra subconsulta:

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

Para obtener más información, vea Realizar combinaciones agrupadas.

Combinación externa izquierda

En una combinación externa izquierda se devuelven todos los elementos de la secuencia de origen izquierda, incluso si no hay elementos coincidentes en la secuencia derecha. Para efectuar una combinación externa izquierda en LINQ, use el método `DefaultIfEmpty` junto con una combinación agrupada para especificar un elemento derecho predeterminado para que se genere si un elemento izquierdo no tiene coincidencias. Puede usar `null` como valor predeterminado para cualquier tipo de referencia, aunque también puede especificar un tipo predeterminado definido por el usuario. En el ejemplo siguiente se muestra un tipo predeterminado definido por el usuario:

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

Para obtener más información, vea Realizar operaciones de combinación externa izquierda.

El operador de igualdad

La cláusula `join` efectúa una combinación de igualdad. En otras palabras, solo puede basar las coincidencias en la igualdad de dos claves. No se admiten otros tipos de comparaciones, como "mayor que" o "no es igual a". Para aclarar que todas las combinaciones son combinaciones de igualdad, la cláusula `join` usa la palabra clave `equals` en lugar del operador `==`. La palabra clave `equals` solo se puede usar en una cláusula `join` y difiere del operador `==` en un aspecto importante. Con `equals`, la clave izquierda usa la secuencia de origen externa, mientras que la clave derecha usa el origen interno. El origen externo solo está en el ámbito del lado izquierdo de `equals`, mientras que la secuencia de origen interna solo está en el ámbito del lado derecho.

Combinaciones de desigualdad

Puede efectuar combinaciones de desigualdad, combinaciones cruzadas y otras operaciones de

combinación personalizadas usando varias cláusulas `from` para introducir nuevas secuencias en una consulta de manera independiente. Para obtener más información, vea Realizar operaciones de combinación personalizadas.

Uniones en las colecciones de objetos frente a las tablas relacionales

En una expresión de consulta LINQ, las operaciones de combinación se efectúan en las colecciones de objetos. Las colecciones de objetos no se pueden "combinar" exactamente igual que dos tablas relacionales. En LINQ, las cláusulas `join` explícitas solo son necesarias cuando dos secuencias de origen no están unidas por ninguna relación. Cuando se trabaja con LINQ to SQL, las tablas de claves externas se representan en el modelo de objetos como propiedades de la tabla principal. Por ejemplo, en la base de datos Northwind, la tabla Customer (Cliente) tiene una relación de clave externa con la tabla Orders (Pedidos). Al asignar las tablas al modelo de objetos, la clase Customer tiene una propiedad Orders que contiene la colección Orders asociada a ese cliente. De hecho, la combinación ya se ha llevado a cabo automáticamente.

Para obtener más información sobre las consultas en tablas relacionadas en el contexto de LINQ to SQL, vea Cómo: Asignar relaciones de base de datos.

Claves compuestas

Puede probar la igualdad de varios valores mediante una clave compuesta. Para obtener más información, vea Realizar una unión usando claves compuestas. Las claves compuestas también se pueden usar en una cláusula `group`.

Ejemplo

En el ejemplo siguiente se comparan los resultados de una combinación interna, una combinación agrupada y una combinación externa izquierda en los mismos orígenes de datos mediante las mismas claves coincidentes. Se ha agregado algún código adicional a estos ejemplos para aclarar los resultados en la pantalla de la consola.

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };
```

```
    // Specify the second data source.
    List<Product> products = new List<Product>()
   {
      new Product {Name="Cola",  CategoryID=001},
      new Product {Name="Tea",  CategoryID=001},
      new Product {Name="Mustard", CategoryID=002},
      new Product {Name="Pickles", CategoryID=002},
      new Product {Name="Carrots", CategoryID=003},
      new Product {Name="Bok Choy", CategoryID=003},
      new Product {Name="Peaches", CategoryID=005},
      new Product {Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    void InnerJoin()
    {
        // Create the query that selects
        // a property from each element.
        var innerJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID
           select new { Category = category.ID, Product = prod.Name };

        Console.WriteLine("InnerJoin:");
        // Execute the query. Access results
        // with a simple foreach statement.
        foreach (var item in innerJoinQuery)
        {
            Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
        }
        Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin()
    {
        // This is a demonstration query to show the output
        // of a "raw" group join. A more typical group join
        // is shown in the GroupInnerJoin method.
        var groupJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           select prodGroup;

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Simple GroupJoin:");

        // A nested foreach statement is required to access group items.
        foreach (var prodGrouping in groupJoinQuery)
        {
            Console.WriteLine("Group:");
```

```
            Console.WriteLine("Group: {0}");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("   {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems,
groupJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.CategoryID into prodGroup
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                           orderby prod2.Name
                           select prod2
            };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine("  {0,-10} {1}", prodItem.Name, prodItem.CategoryID);
            }
        }
        Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems,
groupJoinQuery2.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals product.CategoryID into prodGroup
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("   {0}:{1}", item.ProductName, item.Category);
        }

        Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
```

```
    {
        // Create the query.
        var leftOuterQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID =
category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement  is required to access group items
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems,
leftOuterQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           from item in prodGroup.DefaultIfEmpty()
           select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");

        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
        }
        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.


Unshaped GroupJoin:
Group:
    Cola      1
```

```
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups


GroupInnerJoin:
Beverages
    Cola       1
    Tea        1
Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
    Melons     5
    Peaches    5
GroupInnerJoin: 8 items in 5 named groups


GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group


Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups


LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
```

```
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

Comentarios

Una cláusula `join` que no va seguida de `into` se convierte en una llamada al método Join. Una cláusula `join` que va seguida de `into` se convierte en una llamada al método GroupJoin.

Vea también

- Palabras clave para consultas (LINQ)
- Language-Integrated Query (LINQ)
- Operaciones de combinación
- group (cláusula)
- Realizar operaciones de combinación externa izquierda
- Realizar combinaciones internas
- Realizar combinaciones agrupadas
- Ordenar los resultados de una cláusula join
- Realizar una unión usando claves compuestas
- Sistemas de base de datos compatible para Visual Studio

let (Cláusula, Referencia de C#)

29/06/2020 • 2 minutes to read • Edit Online

En una expresión de consulta, a veces resulta útil almacenar el resultado de una subexpresión para usarlo en las cláusulas siguientes. Puede hacer esto con la palabra clave `let`, que crea una variable de rango y la inicializa con el resultado de la expresión que proporcione. Una vez inicializada con un valor, la variable de rango no se puede usar para almacenar otro valor. En cambio, si la variable de rango contiene un tipo consultable, se puede consultar.

Ejemplo

En el siguiente ejemplo, se usa `let` de dos maneras:

1. Para crear un tipo enumerable que se puede consultar.
2. Para habilitar la consulta para que llame a `ToLower` solo una vez en la variable de rango `word`. Sin usar `let`, tendría que llamar a `ToLower` en cada predicado de la cláusula `where`.

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

Vea también

- Referencia de C#
- Palabras clave para consultas (LINQ)
- LINQ en C#
- Language-Integrated Query (LINQ)
- Controlar excepciones en expresiones de consulta

ascending (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `ascending` se usa en la cláusula orderby en expresiones de consulta para especificar que el criterio de ordenación es de menor a mayor. Como `ascending` es el criterio de ordenación predeterminado, no tiene que especificarlo.

Ejemplo

En el ejemplo siguiente se muestra el uso de `ascending` en una cláusula orderby.

```
IEnumerable<string> sortAscendingQuery =
    from vegetable in vegetables
    orderby vegetable ascending
    select vegetable;
```

Vea también

- Referencia de C#
- LINQ en C#
- descending

descending (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `descending` se usa en la cláusula orderby en expresiones de consulta para especificar que el criterio de ordenación es de mayor a menor.

Ejemplo

En el ejemplo siguiente se muestra el uso de `descending` en una cláusula orderby.

```
IEnumerable<string> sortDescendingQuery =
    from vegetable in vegetables
    orderby vegetable descending
    select vegetable;
```

Vea también

- Referencia de C#
- LINQ en C#
- ascending

on (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `on` se usa en la cláusula join de una expresión de consulta para especificar la condición de combinación.

Ejemplo

En el ejemplo siguiente se muestra el uso de `on` en una cláusula `join`.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

Vea también

- Referencia de C#
- Language-Integrated Query (LINQ)

equals (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `equals` se usa en una cláusula `join` en una expresión de consulta para comparar los elementos de dos secuencias. Para obtener más información, vea join (Cláusula, Referencia de C#).

Ejemplo

En el ejemplo siguiente se muestra el uso de la palabra clave `equals` en una cláusula `join`.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

Vea también

- Language-Integrated Query (LINQ)

by (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave contextual `by` se usa en la cláusula `group` en una expresión de consulta para especificar cómo deben agruparse los elementos devueltos. Para obtener más información, vea group (Cláusula).

Ejemplo

En el ejemplo siguiente se muestra el uso de la palabra clave contextual `by` en una cláusula `group` para especificar que los estudiantes deben agruparse según la primera letra del apellido de cada estudiante.

```
var query = from student in students
            group student by student.LastName[0];
```

Vea también

- LINQ en C#

in (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

La palabra clave `in` se usa en cuatro contextos:

- parámetros de tipo genérico en interfaces y delegados genéricos.
- Como un modificador de parámetro, que le permite pasar un argumento a un método mediante una referencia en lugar de mediante un valor.
- Instrucciones foreach
- Cláusulas from en expresiones de consulta de LINQ.
- Cláusulas join en expresiones de consulta de LINQ

Vea también

- Palabras clave de C#
- Referencia de C#

Operadores de C# (referencia de C#)

04/05/2020 • 6 minutes to read • Edit Online

C# proporciona una serie de operadores compatibles con los tipos integrados. Por ejemplo, los operadores aritméticos realizan operaciones aritméticas con operandos numéricos, y los operadores lógicos booleanos realizan operaciones lógicas con los operandos bool. Algunos operadores se pueden sobrecargar. Con la sobrecarga de operadores, puede especificar el comportamiento del operador para los operandos de un tipo definido por el usuario.

En una expresión, la prioridad y la asociatividad de los operadores determinan el orden en el que se realizan las operaciones. Puede usar los paréntesis para cambiar el orden de evaluación impuesto por la prioridad y la asociatividad de operadores.

Prioridad de operadores

En una expresión con varios operadores, los operadores con mayor prioridad se evalúan antes que los operadores con menor prioridad. En el ejemplo siguiente, la multiplicación se realiza primero porque tiene mayor prioridad que la suma:

```
var a = 2 + 2 * 2;
Console.WriteLine(a); //  output: 6
```

Use paréntesis para cambiar el orden de evaluación impuesto por la prioridad de los operadores:

```
var a = (2 + 2) * 2;
Console.WriteLine(a); //  output: 8
```

En la tabla siguiente se muestran los operadores de C# desde la precedencia más alta a la más baja. Los operadores de cada fila comparten la prioridad.

OPERADORES	CATEGORÍA O NOMBRE
x.y, f(x), a[i], `x?.y`, `x?[y]`, x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Principal
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false	Unario
x..y	Intervalo
switch	Expresión `switch`
x * y, x / y, x % y	Multiplicativo
x + y, x – y	Aditivo
x << y, x >> y	Shift

OPERADORES	CATEGORÍA O NOMBRE
x < y, x > y, x <= y, x >= y, is, as	Comprobación de tipos y relacional
x == y, x != y	Igualdad
`x & y`	Operador lógico booleano AND u operador lógico bit a bit AND
`x ^ y`	Operador lógico booleano XOR u operador lógico bit a bit XOR
`x | y`	Operador lógico booleano OR u operador lógico bit a bit OR
x && y	AND condicional
x || y	OR condicional
x ?? y	Operador de uso combinado de null
c ? t : f	Operador condicional
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y, x ??= y, =>	Asignación y declaración lambda

Asociatividad de los operadores

Cuando los operadores tienen la misma prioridad, su asociatividad determina el orden en el que se realizan las operaciones:

- Los operadores *asociativos a la izquierda* se evalúan, por orden, de izquierda a derecha. A excepción de los operadores de asignación y el operador de integración nula , todos los operadores binarios son asociativos a la izquierda. Por ejemplo, `a + b - c` se evalúa como `(a + b) - c`.
- Los operadores *asociativos a la derecha* se evalúan, por orden, de derecha a izquierda. Los operadores de asignación, el operador de integración nula y el operador condicional `?:` son asociativos a la derecha. Por ejemplo, `x = y = z` se evalúa como `x = (y = z)`.

Use paréntesis, para cambiar el orden de evaluación impuesto por la asociatividad de los operadores:

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}");  // output: a = 1, b = 6
```

Evaluación de operandos

Independientemente de la prioridad y la asociatividad de los operadores, los operandos de una expresión se evalúan de izquierda a derecha. En los siguientes ejemplos, se muestra el orden en el que se evalúan los operadores y los operandos:

EXPRESIÓN	ORDEN DE EVALUACIÓN
`a + b`	a, b, +
`a + b * c`	a, b, c, *, +
`a / b + c * d`	a, b, /, c, d, *, +
`a / (b + c) * d`	a, b, c, +, /, d, *

Normalmente, se evalúan todos los operandos de un operador. Sin embargo, algunos operadores evalúan los operandos de forma condicional. Esto significa que el valor del operando situado más a la izquierda de este tipo de operador define si se deben evaluar otros operandos, o bien qué operandos deben evaluarse. Estos operadores son los operadores lógicos condicionales AND (`&&`) y OR (`||`), los operadores de integración nula `??` y `??=`, los operadores condicionales nulos `?.` y `?[]`, así como el operador condicional `?:`. Para más información, consulte la descripción de cada operador.

Especificación del lenguaje C#

Para obtener más información, vea la sección sobre Operadores de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Expresiones

Operadores aritméticos (referencia de C#)

29/06/2020 • 18 minutes to read • Edit Online

Los operadores siguientes realizan operaciones aritméticas con operandos de tipos numéricos:

- Operadores unarios `++` (incremento), `--` (decremento), `+` (más) y `-` (menos).
- Operadores binarios `*` (multiplicación), `/` (división), `%` (resto), `+` (suma) y `-` (resta).

Estos operadores se admiten en todos los tipos numéricos enteros y de punto flotante.

En el caso de tipos enteros, dichos operadores (excepto los operadores `++` y `--`) se definen para los tipos `int`, `uint`, `long` y `ulong`. Cuando los operandos son de otro tipo entero (`sbyte`, `byte`, `short`, `ushort` o `char`), sus valores se convierten en el tipo `int`, que también es el tipo de resultado de una operación. Cuando los operandos son de distintos tipos enteros o de punto flotante, sus valores se convierten al tipo contenedor más cercano, si ese tipo existe. Para obtener más información, vea la sección Promociones numéricas de Especificación del lenguaje C#. Los operadores `++` y `--` se definen para todos los tipos numéricos enteros y de punto flotante y el tipo char.

Operador de incremento ++

El operador de incremento unario `++` incrementa su operando en 1. El operando debe ser una variable, un acceso de propiedad o un acceso de indexador.

El operador de incremento se admite en dos formas: el operador de incremento posfijo (`x++`) y el operador de incremento prefijo (`++x`).

Operador de incremento de postfijo

El resultado de `x++` es el valor de `x` *antes* de la operación, tal y como se muestra en el ejemplo siguiente:

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4
```

Operador de incremento prefijo

El resultado de `++x` es el valor de `x` *después* de la operación, tal y como se muestra en el ejemplo siguiente:

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

Operador de decremento --

El operador de decremento unario `--` disminuye su operando en 1. El operando debe ser una variable, un acceso de propiedad o un acceso de indexador.

El operador de decremento se admite en dos formas: el operador de decremento posfijo (`x--`) y el operador de decremento prefijo (`--x`).

Operador de decremento de postfijo

El resultado de `x--` es el valor de `x` *antes* de la operación, tal y como se muestra en el ejemplo siguiente:

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2
```

Operador de decremento de prefijo

El resultado de `--x` es el valor de `x` *después* de la operación, tal y como se muestra en el ejemplo siguiente:

```
double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

Operadores unarios más y menos

El operador `+` unario devuelve el valor de su operando. El operador unario `-` calcula la negación numérica del operando.

```
Console.WriteLine(+4);     // output: 4

Console.WriteLine(-4);     // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);            // output: -5
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

El operador unario `-` no es compatible con el tipo ulong.

Operador de multiplicación *

El operador de multiplicación `*` calcula el producto de sus operandos:

```
Console.WriteLine(5 * 2);         // output: 10
Console.WriteLine(0.5 * 2.5);     // output: 1.25
Console.WriteLine(0.1m * 23.4m);  // output: 2.34
```

El operador unario `*` es el operador de direccionamiento indirecto del puntero.

Operador de división /

El operador de división `/` divide el operando izquierdo entre el derecho.

División de enteros

Para los operandos de tipos enteros, el resultado del operador `/` es de un tipo entero y equivale al cociente de los dos operandos redondeados hacia cero:

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

Para obtener el cociente de los dos operandos como número de punto flotante, use el tipo `float`, `double` o `decimal`:

```
Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6
```

División de punto flotante

Para los tipos `float`, `double` y `decimal`, el resultado del operador `/` es el cociente de los dos operandos:

```
Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

Si uno de los operandos es `decimal`, otro operando no puede ser `float` ni `double`, ya que ni `float` ni `double` se convierte de forma implícita a `decimal`. Debe convertir explícitamente el operando `float` o `double` al tipo `decimal`. Para obtener más información sobre las conversiones entre tipos numéricos, consulte Conversiones numéricas integradas.

Operador de resto %

El operador de resto `%` calcula el resto después de dividir el operando izquierdo entre el derecho.

Resto entero

En el caso de los operandos de tipos enteros, el resultado de `a % b` es el valor producido por `a - (a / b) * b`. El signo de resto distinto de cero es el mismo que el del operando izquierdo, como se muestra en el ejemplo siguiente:

```
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

Use el método Math.DivRem para calcular los resultados de la división de enteros y del resto.

Resto de punto flotante

En el caso de los operandos `float` y `double`, el resultado de `x % y` para `x` e `y` finitos es el valor `z`, de modo que

- el signo de `z`, si no es cero, es el mismo que el signo de `x`;
- el valor absoluto de `z` es el valor producido por `|x| - n * |y|`, donde `n` es el entero más grande posible que sea menor o igual que `|x| / |y|`, y `|x|` e `|y|` son los valores absolutos de `x` e `y`, respectivamente.

> **NOTE**
> Este método de cálculo del resto es análogo al que se usa para los operandos enteros, pero difiere de la especificación IEEE 754. Si necesita la operación de resto conforme con la especificación IEEE 754, use el método Math.IEEERemainder.

Para información sobre el comportamiento del operador `%` con operandos no finitos, vea la sección Operador de resto de la Especificación del lenguaje C#.

En el caso de los operandos `decimal`, el operador de resto `%` es equivalente al operador de resto del tipo System.Decimal.

En el ejemplo siguiente se muestra el comportamiento del operador de resto con operandos de punto flotante:

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

Operador de suma +

El operador de suma `+` calcula la suma de sus operandos:

```
Console.WriteLine(5 + 4);       // output: 9
Console.WriteLine(5 + 4.3);     // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

También puede usar el operador `+` para la concatenación de cadenas y la combinación de delegados. Para obtener más información, consulte Operadores `+` y `+=`.

Operador de resta -

El operador de resta `-` resta el operando derecho del izquierdo:

```
Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

También puede usar el operador `-` para la eliminación de delegados. Para obtener más información, consulte Operadores `-` y `-=`.

Asignación compuesta

Para un operador binario `op`, una expresión de asignación compuesta con el formato

```
x op= y
```

es equivalente a

```
x = x op y
```

salvo que `x` solo se evalúa una vez.

En el ejemplo siguiente se muestra el uso de la asignación compuesta con operadores aritméticos:

```
int a = 5;
a += 9;
Console.WriteLine(a);  // output: 14

a -= 4;
Console.WriteLine(a);  // output: 10

a *= 2;
Console.WriteLine(a);  // output: 20

a /= 4;
Console.WriteLine(a);  // output: 5

a %= 3;
Console.WriteLine(a);  // output: 2
```

A causa de las promociones numéricas, el resultado de la operación `op` podría no ser convertible de forma implícita en el tipo `T` de `x`. En tal caso, si `op` es un operador predefinido y el resultado de la operación es convertible de forma explícita en el tipo `T` de `x`, una expresión de asignación compuesta con el formato `x op= y` es equivalente a `x = (T)(x op y)`, excepto que `x` solo se evalúa una vez. En el ejemplo siguiente se muestra ese comportamiento:

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType());  // output: System.Int32
Console.WriteLine(c);  // output: 300

a += b;
Console.WriteLine(a);  // output: 44
```

Los operadores `+=` y `-=` también se usan para suscribirse y cancelar la suscripción a un evento, respectivamente. Para obtener más información, vea Procedimiento para suscribir y cancelar la suscripción a eventos.

Prioridad y asociatividad de los operadores

En la lista siguiente se ordenan los operadores aritméticos de la prioridad más alta a la más baja:

- Operadores de incremento `x++` y decremento `x--` posfijos
- Operadores de incremento `++x` y decremento `--x` prefijos, y operadores unarios `+` y `-`.
- Operadores de multiplicación `*`, `/` y `%`.
- Operadores de suma adición `+` y `-`.

Los operadores aritméticos binarios son asociativos a la izquierda. Es decir, los operadores con el mismo nivel de prioridad se evalúan de izquierda a derecha.

Use los paréntesis, `()`, para cambiar el orden de evaluación impuesto por la prioridad y la asociatividad de operadores.

```
Console.WriteLine(2 + 2 * 2);   // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2);   // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

Para obtener la lista completa de los operadores de C# ordenados por nivel de prioridad, vea la sección Prioridad de operadores del artículo Operadores de C#.

Desbordamiento aritmético y división por cero

Si el resultado de una operación aritmética está fuera del intervalo de valores finitos posibles del tipo numérico implicado, el comportamiento de un operador aritmético depende del tipo de sus operandos.

Desbordamiento aritmético de enteros

La división de enteros por cero siempre produce una DivideByZeroException.

En el caso de desbordamiento aritmético de enteros, el comportamiento resultante lo controla un contexto de comprobación de desbordamiento, que puede ser comprobado o no comprobado:

- En un contexto comprobado, si el desbordamiento se produce en una expresión constante, se produce un error de compilación. De lo contrario, si la operación se realiza en tiempo de ejecución, se inicia una excepción OverflowException.
- En un contexto no comprobado, el resultado se trunca mediante el descarte de los bits de orden superior que no caben en el tipo de destino.

Junto con las instrucciones comprobadas y no comprobadas, puede usar los operadores `checked` y `unchecked` para controlar el contexto de comprobación de desbordamiento, en el que se evalúa una expresión:

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b));  // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

De forma predeterminada, las operaciones aritméticas se producen en un contexto *no comprobado*.

Desbordamiento aritmético de punto flotante

Las operaciones aritméticas con los tipos `float` y `double` nunca inician una excepción. El resultado de las operaciones aritméticas con esos tipos puede ser uno de varios valores especiales que representan infinito y no un número:

```
double a = 1.0 / 0.0;
Console.WriteLine(a);                    // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b);                // output: NaN
Console.WriteLine(double.IsNaN(b));  // output: True
```

Para los operandos del tipo `decimal`, el desbordamiento aritmético siempre inicia una excepción OverflowException y la división por cero siempre inicia una excepción DivideByZeroException.

Errores de redondeo

Debido a las limitaciones generales de la representación de punto flotante de los números reales y la aritmética de punto flotante, es posible que se produzcan errores de redondeo en los cálculos con tipos de punto flotante. Es decir, es posible que el resultado de una expresión difiera del resultado matemático esperado. En el ejemplo siguiente se muestran varios de estos casos:

```
Console.WriteLine(.41f % .2f); // output: 0.00999999

double a = 0.1;
double b = 3 * a;
Console.WriteLine(b == 0.3);   // output: False
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17

decimal c = 1 / 3.0m;
decimal d = 3 * c;
Console.WriteLine(d == 1.0m);  // output: False
Console.WriteLine(d);          // output: 0.9999999999999999999999999999
```

Para más información, vea los comentarios en las páginas de referencia de System.Double, System.Single o System.Decimal.

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario puede sobrecargar los operadores unarios (`++`, `--`, `+` y `-`), los operadores binarios (`*`, `/`, `%`, `+` y `-`) y los operadores aritméticos. Cuando se sobrecarga un operador binario, también se sobrecarga de forma implícita el operador de asignación compuesta correspondiente. Un tipo definido por el usuario no puede sobrecargar de forma explícita un operador de asignación compuesta.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Operadores de incremento y decremento posfijos
- Operadores de incremento y decremento prefijos
- Operador unario más
- Operador unario menos
- Operador de multiplicación
- Operador de división
- Operador de resto
- Operador de suma
- Operador de resta
- Asignación compuesta
- Operadores checked y unchecked
- Promociones numéricas

Vea también

- Referencia de C#
- Operadores de C#
- System.Math
- System.MathF
- Valores numéricos en .NET

Operadores lógicos booleanos (referencia de C#)

11/05/2020 • 13 minutes to read • Edit Online

Los operandos siguientes realizan operaciones lógicas con los operandos bool:

- Operador unario `!` (negación lógica).
- Operadores binarios `&` (AND lógico), `|` (OR lógico) y `^` (OR exclusivo lógico). Esos operadores siempre evalúan ambos operandos.
- Operadores binarios `&&` (AND lógico condicional) y `||` (OR lógico condicional). Esos operadores evalúan el operando derecho solo si es necesario.

En el caso de los operandos de los tipos numéricos enteros, los operadores `&`, `|` y `^` realizan operaciones lógicas bit a bit. Para obtener más información, vea Operadores de desplazamiento y bit a bit.

Operador de negación lógico !

El operador `!` de prefijo unario calcula la negación lógica de su operando. Es decir, genera `true`, si el operando se evalúa como `false`, y `false`, si el operando se evalúa como `true`:

```
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False
```

A partir de C# 8.0, el operador `!` de postfijo unario es un operador que permite un valor NULL.

Operador AND lógico &

El operador `&` calcula el operador AND lógico de sus operandos. El resultado de `x & y` es `true` si `x` y `y` se evalúan como `true`. De lo contrario, el resultado es `false`.

El operador `&` evalúa ambos operandos, incluso aunque el izquierdo se evalúe como `false`, para que el resultado de la operación sea `false` con independencia del valor del operando derecho.

En el ejemplo siguiente, el operando derecho del operador `&` es una llamada de método, que se realiza independientemente del valor del operando izquierdo:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

El operador AND lógico condicional `&&` también calcula el operador AND lógico de sus operandos, pero no evalúa el operando derecho si el izquierdo se evalúa como `false`.

En el caso de los operandos de los tipos numéricos enteros, el operador `&` calcula el AND lógico bit a bit de sus operandos. El operador `&` unario es el operador address-of.

Operador IR exclusivo lógico ^

El operador `^` calcula el operador OR exclusivo lógica, también conocido como el operador XOR lógico, de sus operandos. El resultado de `x ^ y` es `true` si `x` se evalúa como `true` y `y` se evalúa como `false` o `x` se evalúa como `false` y `y` se evalúa como `true`. De lo contrario, el resultado es `false`. Es decir, en los operandos `bool`, el operador `^` calcula el mismo resultado como el operador de desigualdad `!=`.

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

En el caso de los operandos de los tipos numéricos enteros, el operador `^` calcula el AND lógico bit a bit de sus operandos.

Operador lógico OR |

El operador `|` calcula el operador OR lógico de sus operandos. El resultado de `x | y` es `true` si `x` o `y` se evalúan como `true`. De lo contrario, el resultado es `false`.

El operador `|` evalúa ambos operandos, incluso aunque el izquierdo se evalúe como `true`, para que el resultado de la operación sea `true` con independencia del valor del operando derecho.

En el ejemplo siguiente, el operando derecho del operador `|` es una llamada de método, que se realiza independientemente del valor del operando izquierdo:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

El operador OR lógico condicional `||` también calcula el operador OR lógico de sus operandos, pero no evalúa el operando derecho si el izquierdo se evalúa como `true`.

En el caso de los operandos de los tipos numéricos enteros, el operador `|` calcula el OR lógico bit a bit de sus operandos.

Operador AND lógico condicional &&

El operador AND lógico condicional `&&`, también denominado operador AND lógico "de cortocircuito", calcula el operador AND lógico de sus operandos. El resultado de `x && y` es `true` si `x` y `y` se evalúan como `true`. De lo contrario, el resultado es `false`. Si `x` se evalúa como `false`, `y` no se evalúa.

En el ejemplo siguiente, el operando derecho del operador `&&` es una llamada de método, que no se realiza si el operando izquierdo se evalúa como `false`:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

El operador AND lógico `&` también calcula el operador AND lógico de sus operandos, pero siempre evalúa ambos operandos.

Operador OR lógico condicional ||

El operador OR lógico condicional `||`, también denominado operador OR lógico "de cortocircuito", calcula el operador OR lógico de sus operandos. El resultado de `x || y` es `true` si `x` o `y` se evalúan como `true`. De lo contrario, el resultado es `false`. Si `x` se evalúa como `true`, `y` no se evalúa.

En el ejemplo siguiente, el operando derecho del operador `||` es una llamada de método, que no se realiza si el operando izquierdo se evalúa como `true`:

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

El operador OR lógico `|` también calcula el operador OR lógico de sus operandos, pero siempre evalúa ambos operandos.

Operadores lógicos booleanos que aceptan valores NULL

En el caso de los operandos `bool?`, los operadores `&` y `|` admiten la lógica de tres valores. La semántica de

estos operadores se define en la tabla siguiente:

X	Y	X E Y	X|Y
true	true	true	true
true	False	false	true
true	null	null	true
False	true	False	true
False	False	False	False
False	null	False	nulo
null	true	null	true
null	False	False	nulo
null	null	null	null

El comportamiento de esos operadores difiere del comportamiento típico del operador con tipos de valor que aceptan valores NULL. Por lo general, un operador que se define para los operandos de un tipo de valor también se puede usar con los operandos del tipo de valor que acepta valores NULL correspondientes. Este tipo de operador genera `null` si alguno de sus operandos se evalúa como `null`. Sin embargo, los operadores `&` y `|` pueden generar un valor no NULL incluso si uno de los operandos es `null`. Para más información sobre el comportamiento de los operadores con tipos de valor que aceptan valores NULL, consulte la sección Operadores de elevación del artículo Tipos de valor que aceptan valores NULL.

También puede usar los operadores `!` y `^` con los operandos `bool?`, como se muestra en el ejemplo siguiente:

```
bool? test = null;
Display(!test);         // output: null
Display(test ^ false);  // output: null
Display(test ^ null);   // output: null
Display(true ^ null);   // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" : b.Value.ToString());
```

Los operadores lógicos condicionales `&&` y `||` no admiten los operandos `bool?`.

Asignación compuesta

Para un operador binario `op`, una expresión de asignación compuesta con el formato

```
x op= y
```

es equivalente a

```
x = x op y
```

salvo que `x` solo se evalúa una vez.

Los operadores `&`, `|` y `^` admiten la asignación compuesta, como se muestra en el ejemplo siguiente:

```
bool test = true;
test &= false;
Console.WriteLine(test);  // output: False

test |= true;
Console.WriteLine(test);  // output: True

test ^= false;
Console.WriteLine(test);  // output: True
```

> **NOTE**
> Los operadores lógicos condicionales `&&` y `||` no admiten la asignación compuesta.

Prioridad de operadores

En la lista siguiente se ordenan los operadores lógicos desde la prioridad más alta a la más baja:

- Operador de negación lógico `!`
- Operador AND lógico `&`
- Operador OR exclusivo lógico `^`
- Operador OR lógico `|`
- Operador AND lógico condicional `&&`
- Operador OR lógico condicional `||`

Use los paréntesis, `()`, para cambiar el orden de evaluación impuesto por la prioridad de los operadores:

```
Console.WriteLine(true | true & false);   // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) && Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) && Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

Para obtener la lista completa de los operadores de C# ordenados por nivel de prioridad, vea la sección Prioridad de operadores del artículo Operadores de C#.

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario puede sobrecargar los operadores `!`, `&`, `|` y `^`. Cuando se sobrecarga un operador binario, también se sobrecarga de forma implícita el operador de asignación compuesta correspondiente. Un tipo definido por el usuario no puede sobrecargar de forma explícita un operador de asignación compuesta.

Un tipo definido por el usuario no puede sobrecargar los operadores lógicos condicionales `&&` y `||`. Sin embargo, si un tipo definido por el usuario sobrecarga los operadores true y false y el operador `&` o `|` de cierta manera, la operación `&&` o `||`, respectivamente, se puede evaluar para los operandos de ese tipo. Para obtener más información, vea la sección Operadores lógicos condicionales definidos por el usuario de la Especificación del lenguaje C#.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Operador de negación lógico
- Operadores lógicos
- Operadores lógicos condicionales
- Asignación compuesta

Vea también

- Referencia de C#
- Operadores de C#
- Operadores de desplazamiento y bit a bit

Operadores de desplazamiento y bit a bit (referencia de C#)

18/03/2020 • 14 minutes to read • Edit Online

Los operadores siguientes realizan operaciones de desplazamiento o bit a bit con operandos de los tipos numéricos enteros o el tipo char:

- Operador unario `~` (complemento bit a bit)
- Operadores de desplazamiento binarios `<<` (desplazamiento izquierdo) y `>>` (desplazamiento derecho)
- Operadores binarios `&` (AND lógico), `|` (OR lógico) y `^` (OR exclusivo lógico)

Estos operadores se definen para los tipos `int`, `uint`, `long` y `ulong`. Cuando ambos operandos son de otro tipo entero (`sbyte`, `byte`, `short`, `ushort` o `char`), sus valores se convierten en el tipo `int`, que también es el tipo de resultado de una operación. Cuando los operandos son de tipo entero diferente, sus valores se convierten en el tipo entero más cercano que contenga. Para obtener más información, vea la sección Promociones numéricas de Especificación del lenguaje C#.

Los operadores `&`, `|` y `^` también se definen para los operandos de tipo `bool`. Para obtener más información, vea Operadores lógicos booleanos.

Las operaciones de desplazamiento y bit a bit nunca producen desbordamiento y generan los mismos resultados en contextos Checked y Unchecked.

Operador de complemento bit a bit ~

El operador `~` genera un complemento bit a bit de su operando al invertir cada bit:

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

También se puede usar el símbolo `~` para declarar finalizadores. Para obtener más información, vea Finalizadores.

Operador de desplazamiento izquierdo <<

El operador `<<` desplaza el operando izquierdo a la izquierda el número de bits definido por el operando derecho.

La operación de desplazamiento izquierdo descarta los bits de orden superior que están fuera del rango del tipo de resultado y establece las posiciones de bits vacías de orden inferior en cero, como se muestra en el ejemplo siguiente:

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

Dado que los operadores de desplazamiento solo se definen para los tipos `int`, `uint`, `long` y `ulong`, el resultado de una operación siempre contiene al menos 32 bits. Si el operando izquierdo es de otro tipo entero (`sbyte`, `byte`, `short`, `ushort` o `char`), su valor se convierte al tipo `int`, como se muestra en el ejemplo siguiente:

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

Para obtener información sobre cómo el operando derecho del operador `<<` define el recuento de desplazamiento, vea la sección Recuento de desplazamiento de los operadores de desplazamiento.

Operador de desplazamiento derecho >>

El operador `>>` desplaza el operando izquierdo a la derecha el número de bits definido por el operando derecho.

La operación de desplazamiento derecho descarta los bits de orden inferior, como se muestra en el ejemplo siguiente:

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2), 4}");
// Output:
// Before: 1001
// After:    10
```

Las posiciones de bits vacíos de orden superior se establecen basándose en el tipo del operando izquierdo, tal como se indica a continuación:

- Si el operando izquierdo es de tipo `int` o `long`, el operador de desplazamiento a la derecha realiza un desplazamiento *aritmético*: el valor del bit más significativo (el bit de signo) del operando izquierdo se propaga a las posiciones de bits vacíos de orden superior. Es decir, las posiciones de bits vacíos de orden superior se establecen en cero si el operando izquierdo no es negativo y, en caso de serlo, se establecen en uno.

```
int a = int.MinValue;
Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

int b = a >> 3;
Console.WriteLine($"After:  {Convert.ToString(b, toBase: 2)}");
// Output:
// Before: 10000000000000000000000000000000
// After:  11110000000000000000000000000000
```

- Si el operando izquierdo es de tipo `uint` o `ulong`, el operador de desplazamiento a la derecha realiza un desplazamiento *lógico*: las posiciones de bits vacíos de orden superior se establecen siempre en cero.

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After:  {Convert.ToString(d, toBase: 2), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:     10000000000000000000000000000
```

Para obtener información sobre cómo el operando derecho del operador `>>` define el recuento de desplazamiento, vea la sección Recuento de desplazamiento de los operadores de desplazamiento.

Operador AND lógico &

El operador `&` calcula el AND lógico bit a bit de sus operandos:

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

Para los operandos `bool`, el operador `&` calcula el AND lógico de sus operandos. El operador `&` unario es el operador address-of.

Operador IR exclusivo lógico ^

El operador `^` calcula el OR exclusivo lógico bit a bit, también conocido como XOR lógico bit a bit, de sus operandos:

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

Para los operandos `bool`, el operador `^` calcula el OR exclusivo lógico de sus operandos.

Operador lógico OR |

El operador `|` calcula el OR lógico bit a bit de sus operandos:

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

Para los operandos `bool`, el operador `|` calcula el OR lógico de sus operandos.

Asignación compuesta

Para un operador binario `op`, una expresión de asignación compuesta con el formato

```
x op= y
```

es equivalente a

```
x = x op y
```

salvo que `x` solo se evalúa una vez.

En el ejemplo siguiente se muestra el uso de la asignación compuesta con operadores de desplazamiento y bit a bit:

```
uint a = 0b_1111_1000;
a &= 0b_1001_1101;
Display(a);  // output: 10011000

a |= 0b_0011_0001;
Display(a);  // output: 10111001

a ^= 0b_1000_0000;
Display(a);  // output:   111001

a <<= 2;
Display(a);  // output: 11100100

a >>= 4;
Display(a);  // output:     1110

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 8}");
```

A causa de las promociones numéricas, el resultado de la operación `op` podría no ser convertible de forma implícita en el tipo `T` de `x`. En tal caso, si `op` es un operador predefinido y el resultado de la operación es convertible de forma explícita en el tipo `T` de `x`, una expresión de asignación compuesta con el formato `x op= y` es equivalente a `x = (T)(x op y)`, excepto que `x` solo se evalúa una vez. En el ejemplo siguiente se muestra ese comportamiento:

```
byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}");  // output: 1111000100000000

x <<= 8;
Console.WriteLine(x);  // output: 0
```

Prioridad de operadores

En la lista siguiente se ordenan los operadores de desplazamiento y bit a bit desde la prioridad más alta a la más baja:

- Operador de complemento bit a bit `~`
- Operadores de desplazamiento `<<` y `>>`
- Operador AND lógico `&`
- Operador OR exclusivo lógico `^`
- Operador OR lógico `|`

Use los paréntesis, `()`, para cambiar el orden de evaluación impuesto por la prioridad de los operadores:

```
uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1);  // output: 1101

uint d2 = (a | b) & c;
Display(d2);  // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 4}");
```

Para obtener la lista completa de los operadores de C# ordenados por nivel de prioridad, vea la sección Prioridad de operadores del artículo Operadores de C#.

Recuento de desplazamiento de los operadores de desplazamiento

En el caso de los operadores de desplazamiento `<<` y `>>`, el tipo del operando derecho debe ser `int` o un tipo que tenga una conversión numérica implícita predefinida a `int`.

En el caso de las expresiones `x << count` y `x >> count`, el recuento de desplazamiento real depende del tipo de `x`, de esta forma:

- Si el tipo de `x` es `int` o `uint`, el recuento de desplazamiento viene definido por los *cinco* bits de orden inferior del operando derecho. Es decir, el valor de desplazamiento se calcula a partir de `count & 0x1F` (o `count & 0b_1_1111`).
- Si el tipo de `x` es `long` o `ulong`, el recuento de desplazamiento viene definido por los *seis* bits de orden inferior del operando derecho. Es decir, el valor de desplazamiento se calcula a partir de `count & 0x3F` (o `count & 0b_11_1111`).

En el ejemplo siguiente se muestra ese comportamiento:

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2
```

NOTE

Tal y como se muestra en el ejemplo anterior, el resultado de una operación de desplazamiento puede ser distinto de cero incluso si el valor del operando derecho es mayor que el número de bits del operando izquierdo.

Operadores lógicos de enumeración

Los operadores `~`, `&`, `|` y `^` también se admiten en cualquier tipo de enumeración. En el caso de los operandos del mismo tipo de enumeración, se realiza una operación lógica en los valores correspondientes del tipo entero subyacente. Por ejemplo, para cualquier `x` e `y` de un tipo de enumeración `T` con un tipo subyacente `U`, la expresión `x & y` produce el mismo resultado que la expresión `(T)((U)x & (U)y)`.

Normalmente, los operadores lógicos bit a bit se usan con un tipo de enumeración definido con el atributo Flags. Para obtener más información, vea la sección Tipos de enumeración como marcas de bits del artículo Tipos de enumeración.

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario puede sobrecargar los operadores `~`, `<<`, `>>`, `&`, `|` y `^`. Cuando se sobrecarga un operador binario, también se sobrecarga de forma implícita el operador de asignación compuesta correspondiente. Un tipo definido por el usuario no puede sobrecargar de forma explícita un operador de asignación compuesta.

Si un tipo definido por el usuario `T` sobrecarga el operador `<<` o `>>`, el tipo del operando izquierdo debe ser `T` y el del derecho `int`.

especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Operador de complemento bit a bit
- Operadores de desplazamiento
- Operadores lógicos
- Asignación compuesta
- Promociones numéricas

Vea también

- Referencia de C#
- Operadores de C#
- Operadores lógicos booleanos

Operadores de igualdad (referencia de C#)

02/04/2020 • 6 minutes to read • Edit Online

Los operadores `==` (igualdad) y `!=` (desigualdad) comprueban si los operandos son iguales.

Operador de igualdad ==

El operador de igualdad `==` devuelve `true` si sus operandos son iguales; en caso contrario, devuelve `false`.

Igualdad entre tipos de valor

Los operandos de los tipos de valor integrados son iguales si sus valores son iguales:

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True
```

NOTE

Para los operadores `==`, `<`, `>`, `<=` y `>=`, si cualquier operando no es un número (Double.NaN o Single.NaN), el resultado del operador será `false`. Esto significa que el valor `NaN` no es mayor, inferior ni igual que cualquier otro valor `double` o `float`, incluido `NaN`. Para obtener más información y ejemplos, vea el artículo de referencia Double.NaN o Single.NaN.

Dos operandos del mismo tipo enum son iguales si los valores correspondientes del tipo entero subyacente son iguales.

Los tipos struct definidos por el usuario no son compatibles con el operador `==` de forma predeterminada. Para admitir el operador `==`, un elemento struct definido por el usuario debe sobrecargarlo.

A partir de C# 7.3, los operadores `==` y `!=` son compatibles con las tuplas de C#. Para obtener más información, vea la sección Igualdad y tuplas del artículo Tipos de tupla de C#.

Igualdad entre tipos de referencia

De forma predeterminada, dos operandos de tipo de referencia son iguales si hacen referencia al mismo objeto:

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b);  // output: False
        Console.WriteLine(a == c);  // output: True
    }
}
```

Como se muestra en el ejemplo, el operador `==` es compatible de forma predeterminada con los tipos de referencia definidos por el usuario. Sin embargo, un tipo de referencia puede sobrecargar el operador `==`. Si un tipo de referencia sobrecarga el operador `==`, use el método Object.ReferenceEquals para comprobar si dos referencias de ese tipo hacen referencia al mismo objeto.

Igualdad entre cadenas

Dos operandos string son iguales si ambos son `null`, o bien si las instancias de ambas cadenas tienen la misma longitud y los mismos caracteres en cada posición de caracteres:

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```

Es la comparación de ordinales que distingue entre mayúsculas de minúsculas. Para obtener más información sobre la comparación de cadenas, vea Cómo comparar cadenas en C# .

Igualdad entre delegados

Dos operandos de delegado con el mismo tipo de entorno de ejecución son iguales cuando ambos son `null`, o bien cuando sus listas de invocación tienen la misma longitud y las mismas entradas en cada posición:

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c));  // output: False
Console.WriteLine(b == c);  // output: True
```

Para obtener más información, vea la sección sobre los operadores de igualdad entre delegados de la Especificación del lenguaje C#.

Los delegados que se producen mediante la evaluación de expresiones lambda semánticamente idénticas no son iguales, tal como se muestra en el ejemplo siguiente:

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b);  // output: False
Console.WriteLine(a + b == a + b);  // output: True
Console.WriteLine(b + a == a + b);  // output: False
```

Operador de desigualdad !=

El operador de desigualdad (`!=`) devuelve `true` si sus operandos son iguales; en caso contrario, devuelve `false`. Para los operandos de los tipos integrados, la expresión `x != y` genera el mismo resultado que la expresión `!(x == y)`. Para obtener más información sobre la igualdad de tipos, vea la sección Operador de igualdad.

En el siguiente ejemplo se muestra el uso del operador `!=`:

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b);  // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2);  // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario puede sobrecargar los operadores `==` y `!=`. Si un tipo sobrecarga uno de los dos operadores, también debe sobrecargar el otro.

Especificación del lenguaje C#

Para obtener más información, vea la sección Operadores de comprobación de tipos y relacionales de la Especificación del lenguaje de C#.

Vea también

- Referencia de C#
- Operadores de C#
- System.IEquatable<T>
- Object.Equals
- Object.ReferenceEquals
- Comparaciones de igualdad
- Operadores de comparación

Operadores de comparación (referencia de C#)

29/06/2020 • 3 minutes to read • Edit Online

Los operadores de la comparación `<` (menor que), `>` (mayor que), `<=` (menor o igual que) y `>=` (mayor o igual que), también conocidos como relacionales, comparan sus operandos. Estos operadores se admiten en todos los tipos numéricos enteros y de punto flotante.

> **NOTE**
>
> Para los operadores `==`, `<`, `>`, `<=` y `>=`, si cualquier operando no es un número (Double.NaN o Single.NaN), el resultado del operador será `false`. Esto significa que el valor `NaN` no es mayor, inferior ni igual que cualquier otro valor `double` o `float`, incluido `NaN`. Para obtener más información y ejemplos, vea el artículo de referencia Double.NaN o Single.NaN.

El tipo char también admite operadores de comparación. En el caso de los operandos `char`, se comparan los códigos de caracteres correspondientes.

Los tipos de enumeración también admiten operadores de comparación. Para los operandos del mismo tipo enum, se comparan los valores correspondientes del tipo entero subyacente.

Los operadores `==` y `!=` comprueban si los operandos son iguales.

Operador menor que <

El operador `<` devuelve `true` si el operando izquierdo es menor que el derecho; en caso contrario, devuelve `false`.

```
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Operador mayor que >

El operador `>` devuelve `true` si el operando izquierdo es mayor que el derecho; en caso contrario, devuelve `false`.

```
Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False
Console.WriteLine(0.0 > 5.1);   // output: False

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Operador menor o igual que <=

El operador `<=` devuelve `true` si el operando izquierdo es menor o igual que el derecho; en caso contrario, devuelve `false`.

```
Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(0.0 <= 5.1);   // output: True

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Operador mayor o igual que >=

El operador `>=` devuelve `true` si el operando izquierdo es mayor o igual que el derecho; en caso contrario, devuelve `false`.

```
Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True
Console.WriteLine(0.0 >= 5.1);   // output: False

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario puede sobrecargar los operadores `<`, `>`, `<=` y `>=`.

Si un tipo sobrecarga uno de los operadores `<` o `>`, también debe sobrecargar `<` y `>`. Si un tipo sobrecarga uno de los operadores `<=` o `>=`, también debe sobrecargar `<=` y `>=`.

Especificación del lenguaje C#

Para obtener más información, vea la sección Operadores de comprobación de tipos y relacionales de la Especificación del lenguaje de C#.

Vea también

- Referencia de C#
- Operadores de C#
- System.IComparable<T>
- Operadores de igualdad

Operadores y expresiones de acceso a miembros (referencia de C#)

29/06/2020 • 15 minutes to read • Edit Online

Puede usar los operadores y expresiones siguientes cuando accede a un miembro de tipo:

- `.` (acceso a miembros): para acceder a un miembro de un espacio de nombres o un tipo
- `[]` (elemento de matriz o acceso a indizador): para acceder a un elemento de matriz o un indizador de tipo
- `?.` y `?[]` (operadores condicionales NULL): para realizar una operación de acceso a elementos o miembros solo si un operando es distinto de NULL
- `()` (invocación): para llamar a un método de acceso o invocar un delegado
- `^` (índice desde el final) : para indicar que la posición del elemento se encuentra a partir del final de una secuencia
- `..` (intervalo): para especificar un intervalo de índices que puede utilizar para obtener un intervalo de elementos de secuencia

Expresión de acceso a miembros.

Use el token `.` para acceder a un miembro de un espacio de nombres o un tipo, como se muestran en los ejemplos siguientes:

- Use `.` para acceder a un espacio de nombres anidado dentro de un espacio de nombres, como se muestra en el siguiente ejemplo de una directiva `using`:

  ```
  using System.Collections.Generic;
  ```

- Use `.` para formar un *nombre completo* para tener acceso a un tipo dentro de un espacio de nombres, como se muestra en el código siguiente:

  ```
  System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
  ```

  Utilice una directiva `using` para hacer que el uso de nombres completos sea opcional.

- Use `.` para tener acceso a los miembros de tipo, que no son estáticos y, como se muestra en el código siguiente:

  ```
  var constants = new List<double>();
  constants.Add(Math.PI);
  constants.Add(Math.E);
  Console.WriteLine($"{constants.Count} values to show:");
  Console.WriteLine(string.Join(", ", constants));
  // Output:
  // 2 values to show:
  // 3.14159265358979, 2.71828182845905
  ```

También puede usar `.` para acceder a un método de extensión.

Operador del indizador []

Los corchetes, `[]`, se suelen usar para el acceso a matriz, indizador o elemento de puntero.

Acceso a matriz

En el ejemplo siguiente se muestra cómo se obtiene acceso a los elementos de matriz:

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]);  // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant);  // output: -3
```

Si un índice de matriz se encuentra fuera de los límites de la dimensión correspondiente de una matriz, se produce una excepción IndexOutOfRangeException.

Tal como se muestra en el ejemplo anterior, también usa corchetes al declarar un tipo de matriz o crear instancias de matriz.

Para obtener más información sobre las matrices, consulte Matrices.

Acceso a indizador

En el ejemplo siguiente se usa el tipo Dictionary<TKey,TValue> de .NET para mostrar el acceso al indizador:

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]);  // output: 4.14159265358979
```

Los indizadores le permiten indizar las instancias de un tipo definido por el usuario de un modo similar a la indización de matrices. A diferencia de los índices de matriz, que deben ser enteros, los parámetros de indizador se pueden declarar para ser de cualquier tipo.

Para más información sobre los indizadores, consulte Indizadores.

Otros usos de []

Para información sobre el acceso de los elementos de puntero, consulte la sección Operador de acceso de elemento de puntero del artículo Operadores relacionados con el puntero.

También usa los corchetes para especificar atributos:

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

Operadores condicionales NULL ?. y ?[]

Disponible en C# 6 y versiones posteriores, un operador condicional NULL aplica una operación de acceso a miembros, `?.`, o acceso a elementos, `?[]`, a su operando solo si dicho operando se evalúa como no `null`. Es decir:

- Si `a` se evalúa como `null`, el resultado de `a?.x` o `a?[x]` es `null`.
- Si `a` se evalúa como no NULL, el resultado de `a?.x` o `a?[x]` es el mismo que el resultado de `a.x` o `a[x]`, respectivamente.

> **NOTE**
> Si `a.x` o `a[x]` producen una excepción, `a?.x` o `a?[x]` produciría la misma excepción para `a` no NULL. Por ejemplo, si `a` es una instancia de matriz que no es NULL y `x` está fuera de los límites de `a`, `a?[x]` produciría una excepción IndexOutOfRangeException.

Los operadores de condición NULL se cortocircuitan. Es decir, si una operación en una cadena de la operación de acceso a elementos o miembros condicional devuelve `null`, no se ejecuta el resto de la cadena. En el ejemplo siguiente, `B` no se evalúa si `A` se evalúa como `null` y `C` no se evalúa si `A` o `B` se evalúan como `null`:

```
A?.B?.Do(C);
A?.B?[C];
```

En el siguiente ejemplo se muestra el uso de los operadores `?.` y `?[]`:

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1);  // output: NaN

var numberSets = new List<double[]>
{
    new[] { 1.0, 2.0, 3.0 },
    null
};

var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2);  // output: 6

var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3);  // output: NaN
```

En el ejemplo anterior también se usa el operador de fusión de NULL `??` para especificar una expresión alternativa que se evaluará en caso de que el resultado de la operación condicional NULL sea `null`.

Si `a.x` o `a[x]` es de un tipo de valor que no admite un valor NULL, `T`, `a?.x` o `a?[x]` es del tipo de valor que admite un valor NULL `T?` correspondiente. Si necesita una expresión de tipo `T`, aplique el operador de fusión de NULL `??` a una expresión condicional NULL, tal como se muestra en el ejemplo siguiente:

```
int GetSumOfFirstTwoOrDefault(int[] numbers)
{
    if ((numbers?.Length ?? 0) < 2)
    {
        return 0;
    }
    return numbers[0] + numbers[1];
}

Console.WriteLine(GetSumOfFirstTwoOrDefault(null));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0]));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 }));  // output: 7
```

En el ejemplo anterior, si no utiliza el operador `??`, `numbers?.Length < 2` da como resultado `false` cuando `numbers` es `null`.

El operador de acceso de miembro condicional NULL `?.` también se conoce con el nombre de operador Elvis.

> **NOTE**
> En C# 8, el operador null-forgiving finaliza la lista de operaciones null-conditional anteriores. Por ejemplo, la expresión `x?.y!.z` se analiza como `(x?.y)!.z`. Debido a esta interpretación, `z` se evalúa incluso si `x` es `null`, lo que puede dar lugar a NullReferenceException.

Invocación de delegado seguro para subprocesos

Use el operador `?.` para comprobar si un delegado es distinto de NULL y se invoca de forma segura para subprocesos (por ejemplo, cuando se genera un evento), tal como se muestra en el código siguiente:

```
PropertyChanged?.Invoke(…)
```

Ese código es equivalente al código siguiente que se usaría en C# 5 o una versión anterior:

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(…);
}
```

Es un modo seguro para subprocesos de asegurarse de que solo se invoca un `handler` que no es NULL. Dado que las instancias de delegado son inmutables, ningún subproceso puede cambiar el valor al que hace referencia la variable local `handler`. En concreto, si el código que ha ejecutado otro subproceso cancela la suscripción del evento `PropertyChanged` y `PropertyChanged` se convierte en `null` antes de que se invoque `handler`, el objeto al que hace referencia `handler` queda intacto. El operador `?.` evalúa el operando de la izquierda no más de una vez, lo que garantiza que no se pueda cambiar a `null` después de verificarse como no NULL.

Expresión de invocación ()

Utilice paréntesis, `()`, para llamar a un método o invocar un delegado.

En el ejemplo siguiente se muestra cómo llamar a un método, con o sin argumentos, y cómo invocar un delegado:

```
Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count);   // output: 2

numbers.Clear();
display(numbers.Count);   // output: 0
```

También usa paréntesis al invocar un constructor con el operador `new`.

Otros usos de ()

También usa los paréntesis para ajustar el orden en el que se van a evaluar operaciones en una expresión. Para obtener más información, vea Operadores de C# (referencia de C#).

Expresiones de conversión, que realizan conversiones de tipo explícitas, también utilizan paréntesis.

Indexación desde el operador final ^

Disponible en C# 8.0 y versiones posteriores, el operador `^` indica la posición del elemento a partir del final de una secuencia. En el caso de una secuencia de longitud `length`, `^n` apunta al elemento con desplazamiento `length - n` desde el inicio de una secuencia. Por ejemplo, `^1` apunta al último elemento de una secuencia y `^length` apunta al primer elemento de una secuencia.

```
int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last);  // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast);  // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first);  // output: T
```

Como se muestra en el ejemplo anterior, la expresión `^e` es del tipo System.Index. En la expresión `^e`, el resultado de `e` debe poderse convertir implícitamente a `int`.

También puede usar el operador `^` con el operador de intervalo para crear un intervalo de índices. Para más información, consulte Índices y rangos.

Operador de intervalo .

Disponible en C# 8.0 y versiones posteriores, el operador `..` especifica el inicio y el final de un intervalo de índices como sus operandos. El operando izquierdo es un inicio *inclusivo* de un intervalo. El operando derecho es un inicio *exclusivo* de un intervalo. Cualquiera de los operandos puede ser un índice desde el inicio o desde el final de una secuencia, tal y como muestra el ejemplo siguiente:

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset);  // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner);  // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end);  // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

Como se muestra en el ejemplo anterior, la expresión `a..b` es del tipo System.Range. En la expresión `a..b`, los resultados de `a` y `b` deben poderse convertir implícitamente a `int` o Index.

Puede omitir cualquiera de los operandos del operador `..` para obtener un intervalo abierto:

- `a..` es equivalente a `a..^0`
- `..b` es equivalente a `0..b`
- `..` es equivalente a `0..^0`

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf);  // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf);  // output: 0 10 20

int[] all = numbers[..];
Display(all);  // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

Para más información, consulte Índices y rangos.

Posibilidad de sobrecarga del operador

Los operadores `.`, `()`, `^` y `..` no se pueden sobrecargar. El operador `[]` también se considera un operador que no se puede sobrecargar. Use indizadores para admitir la indización con tipos definidos por el usuario.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Acceso a miembros
- Acceso a elementos
- Operador condicional de NULL
- Expresiones de invocación

Para más información sobre índices y rangos, vea la nota de propuesta de características.

Vea también

- Referencia de C#
- Operadores de C#
- ?? (operador de uso combinado de NULL)
- Operador ::

Operadores de prueba de tipos y expresión de conversión (referencia de C#)

04/05/2020 • 11 minutes to read • Edit Online

Puede usar los siguientes operadores y expresiones para realizar la comprobación de tipos o la conversión de tipos:

- operador is: para comprobar si el tipo en tiempo de ejecución de una expresión es compatible con un tipo determinado.
- operador as: para convertir explícitamente una expresión a un tipo determinado si su tipo en tiempo de ejecución es compatible con ese tipo.
- expresión Cast: para realizar una conversión explícita.
- operador typeof: para obtener la instancia System.Type para un tipo.

Operador is

El operador `is` comprueba si el tipo en tiempo de ejecución del resultado de una expresión es compatible con un tipo determinado. A partir de C# 7.0, el operador `is` también prueba el resultado de una expresión en relación con un patrón.

La expresión con el operador `is` de prueba de tipos tiene el formato siguiente:

```
E is T
```

donde `E` es una expresión que devuelve un valor y `T` es el nombre de un tipo o un parámetro de tipo. `E` no puede ser un método anónimo ni una expresión lambda.

La expresión `E is T` devuelve `true` si el resultado de `E` es distinto de NULL y se puede convertir al tipo `T` por una conversión de referencia, una conversión boxing o una conversión unboxing; de lo contrario, devuelve `false`. El operador `is` no toma en consideración las conversiones definidas por el usuario.

En el ejemplo siguiente se muestra que el operador `is` devuelve `true` si el tipo en tiempo de ejecución del resultado de una expresión se deriva de un tipo determinado, es decir, existe una conversión de referencia entre tipos:

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);  // output: True
        Console.WriteLine(b is Derived);  // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);  // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

En el ejemplo siguiente se muestra que el operador `is` tiene en cuenta las conversiones boxing y unboxing pero no considera las conversiones numéricas:

```
int i = 27;
Console.WriteLine(i is System.IFormattable);  // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int);  // output: True
Console.WriteLine(iBoxed is long);  // output: False
```

Para obtener información acerca de las conversiones de C#, vea el capítulo Conversiones de la especificación del lenguaje C#.

Prueba de tipos con coincidencia de patrones

A partir de C# 7.0, el operador `is` también prueba el resultado de una expresión en relación con un patrón. En concreto, admite el patrón de tipo de la forma siguiente:

```
E is T v
```

donde `E` es una expresión que devuelve un valor, `T` es el nombre de un tipo o un parámetro de tipo y `v` es una nueva variable local de tipo `T`. Si el resultado de `E` no es NULL y puede convertirse en `T` mediante una conversión de referencia, boxing o unboxing, la expresión `E is T v` devuelve `true` y el valor convertido del resultado de `E` se asigna a la variable `v`.

En el siguiente ejemplo se muestra el uso del operador `is` con un patrón de tipo:

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b);  // output 30
}
```

Para obtener más información sobre el patrón de tipo y otros patrones admitidos, consulte Coincidencia de patrones con is.

Operador as

El operador `as` convierte explícitamente el resultado de una expresión en una referencia determinada o un tipo de valor que acepta valores NULL. Si la conversión no es posible, el operador `as` devuelve `null`. A diferencia de la expresión Cast, el operador `as` no genera nunca una excepción.

La expresión con el formato

```
E as T
```

donde `E` es una expresión que devuelve un valor y `T` es el nombre de un tipo o un parámetro de tipo produce el mismo resultado que

```
E is T ? (T)(E) : (T)null
```

salvo que `E` solo se evalúa una vez.

El operador `as` solo considera las conversiones de referencia, las que aceptan valores NULL, boxing y unboxing. No puede usar el operador `as` para realizar una conversión definida por el usuario. Para ello, use una expresión Cast.

En el siguiente ejemplo se muestra el uso del operador `as`:

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]);  // output: 40
}
```

> **NOTE**
> Como se muestra en el ejemplo anterior, se necesita comparar el resultado de la expresión `as` con `null` para comprobar si una conversión es correcta. A partir de C# 7.0, puede usar el operador is para probar si la conversión es correcta y, si es así, asignar su resultado a una nueva variable.

Expresión Cast

Una expresión de conversión con el formato `(T)E` realiza una conversión explícita del resultado de la expresión `E` al tipo `T`. Si no existe ninguna conversión explícita del tipo de `E` al tipo `T`, se producirá un error en tiempo de compilación. En el tiempo de ejecución, una conversión explícita podría no completarse correctamente y una expresión de conversión podría generar una excepción.

El ejemplo siguiente muestra las conversiones explícitas numérica y de referencia:

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a);   // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count);  // output: 3
Console.WriteLine(list[1]);  // output: 20
```

Para obtener más información sobre las conversiones explícitas, vea la sección Conversiones explícitas de la especificación del lenguaje C#. Para obtener información sobre cómo definir una conversión personalizada de tipo

explícito o implícito, vea Operadores de conversión definidos por el usuario.

Otros usos de ()

También puede utilizar paréntesis para llamar a un método o invocar un delegado.

Sirven además para ajustar el orden en el que se van a evaluar operaciones en una expresión. Para obtener más información, vea Operadores de C# (referencia de C#).

typeof (operador)

El operador `typeof` obtiene la instancia System.Type para un tipo. El argumento del operador `typeof` debe ser el nombre de un tipo o un parámetro de tipo, como se muestra en el ejemplo siguiente:

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

También se puede usar el operador `typeof` con tipos genéricos sin enlazar. El nombre de un tipo genérico sin enlazar debe contener el número apropiado de comas, que es inferior en una unidad al número de parámetros de tipo. En el siguiente ejemplo se muestra el uso del operador `typeof` con un tipo genérico sin enlazar:

```
Console.WriteLine(typeof(Dictionary<,>));
// Output:
// System.Collections.Generic.Dictionary`2[TKey,TValue]
```

Una expresión no puede ser un argumento del operador `typeof`. Para obtener la instancia de System.Type para el tipo en tiempo de ejecución del resultado de una expresión, use el método Object.GetType.

Prueba de tipos con el operador `typeof`

Use el operador `typeof` para comprobar si el tipo en tiempo de ejecución del resultado de la expresión coincide exactamente con un tipo determinado. En el ejemplo siguiente se muestra la diferencia entre la comprobación de tipos realizada con el operador `typeof` y el operador is:

```
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal);  // output: True
        Console.WriteLine(b.GetType() == typeof(Animal));  // output: False

        Console.WriteLine(b is Giraffe);  // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe));  // output: True
    }
}
```

Posibilidad de sobrecarga del operador

Los operadores `is`, `as` y `typeof` no se pueden sobrecargar.

Un tipo definido por el usuario no se puede sobrecargar el operador `()`, pero puede definir conversiones de tipos personalizadas que pueden realizarse mediante una expresión de conversión. Para obtener más información, vea Operadores de conversión definidos por el usuario.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- El operador is
- El operador as
- Expresiones de conversión
- El operador typeof

Vea también

- Referencia de C#
- Operadores de C#
- Procedimiento para convertir de forma segura mediante la coincidencia de patrones y los operadores is y as
- Elementos genéricos en .NET

Operadores de conversión definidos por el usuario (referencia de C#)

16/04/2020 • 2 minutes to read • Edit Online

Un tipo definido por el usuario puede definir una conversión implícita o explícita personalizada desde o a otro tipo.

Las conversiones implícitas no requieren que se invoque una sintaxis especial y pueden producirse en diversas situaciones, por ejemplo, en las invocaciones de métodos y asignaciones. Las conversiones implícitas predefinidas en C# siempre se realizan correctamente y nunca inician una excepción. Las conversiones implícitas definidas por el usuario deben comportarse de esta manera. Si una conversión personalizada puede producir una excepción o perder información, se define como una conversión explícita.

Los operadores is y as no tienen en cuenta las conversiones definidas por el usuario. Use una expresión Cast para invocar una conversión explícita definida por el usuario.

Use las palabras clave `operator` y `implicit` o `explicit` para definir una conversión implícita o explícita, respectivamente. El tipo que define una conversión debe ser un tipo de origen o un tipo de destino de dicha conversión. Una conversión entre dos tipos definidos por el usuario se puede definir en cualquiera de los dos tipos.

En el ejemplo siguiente se muestra cómo se define una conversión implícita y explícita:

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number);  // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit);  // output: 7
    }
}
```

Use también la palabra clave `operator` para sobrecargar un operador predefinido en C#. Para obtener más información, vea Sobrecarga de operadores.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Operadores de conversión
- Conversiones definidas por el usuario
- Conversiones implícitas
- Conversiones explícitas

Vea también

- Referencia de C#
- Operadores de C#
- Sobrecarga de operadores
- Operadores de conversión y prueba de tipos
- Conversiones de tipos
- Directrices de diseño: operadores de conversión
- Chained user-defined explicit conversions in C# (Conversiones explícitas encadenadas definidas por el usuario en C#)

Operadores relacionados con el puntero (referencia de C#)

04/05/2020 • 13 minutes to read • Edit Online

Puede usar los siguientes operadores para trabajar con punteros:

- Operador unario `&` (address-of): para obtener la dirección de una variable
- Operador unario `*` (direccionamiento indirecto del puntero): para obtener la variable a la que apunta un puntero
- Operadores `->` (acceso a miembros) y `[]` (acceso de elemento)
- Operadores aritméticos `+`, `-`, `++` y `--`
- Operadores de comparación `==`, `!=`, `<`, `>`, `<=` y `>=`

Para obtener información sobre los tipos de punteros, vea Tipos de puntero.

> **NOTE**
> Cualquier operación con punteros requiere un contexto unsafe. El código que contiene bloques no seguros debe compilarse con la opción del compilador `-unsafe`.

Operador Address-of &

El operador unario `&` devuelve la dirección de su operando:

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(long)pointerToNumber:X}");
}
// Output is similar to:
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

El operando del operador `&` debe ser una variable fija. Las variables *fijas* son las que residen en ubicaciones de almacenamiento que no se ven afectadas por el funcionamiento del recolector de elementos no utilizados. En el ejemplo anterior, la variable local `number` es una variable fija, ya que reside en la pila. Las variables que residen en ubicaciones de almacenamiento que pueden verse afectadas por el recolector de elementos no utilizados (por ejemplo, reubicadas) se denominan variables *móviles*. Los campos de objeto y los elementos de matriz son ejemplos de variables móviles. Puede obtener la dirección de una variable móvil si la "fija" o "ancla" con una instrucción `fixed`. La dirección obtenida solo es válida dentro del bloque de una instrucción `fixed`. En el ejemplo siguiente se muestra cómo usar una instrucción `fixed` y el operador `&`:

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

No se puede obtener la dirección de una constante o un valor.

Para obtener más información sobre las variables fijas y móviles, vea la sección Variables fijas y móviles de Especificación del lenguaje C#.

El operador binario `&` calcula el AND lógico de sus operandos booleanos o el AND lógico bit a bit de sus operandos enteros.

Operador de direccionamiento indirecto del puntero *

El operador unario de direccionamiento indirecto del puntero `*` obtiene la variable a la que apunta su operando. También se conoce como operador de desreferencia. El operando del operador `*` debe ser un tipo de puntero.

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable: {(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update: {letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

No se puede aplicar el operador `*` a una expresión de tipo `void*`.

El operador binario `*` calcula la multiplicación de sus operandos numéricos.

Operador de acceso a miembros de puntero ->

El operador `->` combina el direccionamiento indirecto del puntero y el acceso a miembros. Es decir, si `x` es un puntero de tipo `T*` y `y` es un miembro accesible de tipo `T`, una expresión con el formato

```
x->y
```

es equivalente a

```
(*x).y
```

En el siguiente ejemplo se muestra el uso del operador `->`:

```
public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString());  // output: (3, 4)
    }
}
```

No se puede aplicar el operador `->` a una expresión de tipo `void*`.

Operador de acceso de elemento de puntero []

En el caso de una expresión `p` de un tipo de puntero, un acceso de elemento de puntero con el formato `p[n]` se evalúa como `*(p + n)`, donde `n` debe ser de un tipo convertible de forma implícita en `int`, `uint`, `long` o `ulong`. Para obtener información sobre el comportamiento del operador `+` con punteros, vea la sección Suma o resta de un valor entero en un puntero.

En el ejemplo siguiente se muestra cómo acceder a los elementos de matriz con un puntero y el operador `[]`:

```
unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

En el ejemplo anterior, una expresión `stackalloc` asigna un bloque de memoria en la pila.

> **NOTE**
>
> El operador de acceso de elemento de puntero no busca errores fuera de límites.

No puede usar `[]` para el acceso de elemento de puntero con una expresión de tipo `void*`.

También puede usar el operador `[]` para acceso de elemento de matriz o indizador.

Operadores aritméticos de puntero

Puede realizar las siguientes operaciones aritméticas con punteros:

- Agregar o restar un valor entero en un puntero
- Restar dos punteros
- Incrementar o reducir un puntero

No es posible realizar esas operaciones con punteros de tipo `void*`.

Para obtener información sobre las operaciones aritméticas admitidas con tipos numéricos, vea Operadores aritméticos.

Suma o resta de un valor entero en un puntero

En el caso de un puntero `p` de tipo `T*` y una expresión `n` de un tipo convertible de forma implícita en `int`, `uint`, `long` o `ulong`, la suma y la resta se definen de este modo:

- Ambas expresiones `p + n` y `n + p` generan un puntero de tipo `T*` que resulta de agregar `n * sizeof(T)` a la dirección proporcionada por `p`.
- La expresión `p - n` genera un puntero de tipo `T*` que resulta de restar `n * sizeof(T)` a la dirección proporcionada por `p`.

El operador `sizeof` obtiene el tamaño de un tipo en bytes.

En el siguiente ejemplo se muestra el uso del operador `+` con un puntero:

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address {(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address {(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

Resta de puntero

En el caso de dos punteros `p1` y `p2` de tipo `T*`, la expresión `p1 - p2` genera la diferencia entre las direcciones proporcionadas por `p1` y `p2` dividida por `sizeof(T)`. El tipo del resultado es `long`. Es decir, `p1 - p2` se calcula como `((long)(p1) - (long)(p2)) / sizeof(T)`.

El ejemplo siguiente muestra la resta de puntero:

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1);  // output: 4
}
```

Incremento y decremento de puntero

El operador de incremento `++` agrega 1 a su operando de puntero. El operador de decremento `--` resta 1 a su operando de puntero.

Ambos operadores se admiten con dos formatos: postfijo (`p++` y `p--`) y prefijo (`++p` y `--p`). El resultado de `p++` y `p--` es el valor de `p` *antes* de la operación. El resultado de `++p` y `--p` es el valor de `p` *después* de la operación.

El ejemplo siguiente muestra el comportamiento de los operadores de incremento postfijo y prefijo:

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 - {(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

Operadores de comparación de puntero

Puede usar los operadores `==`, `!=`, `<`, `>`, `<=` y `>=` para comparar los operandos de cualquier tipo de puntero, incluido `void*`. Esos operadores comparan las direcciones proporcionadas por los dos operandos como si fueran enteros sin signo.

Para obtener información sobre el comportamiento de esos operadores para operandos de otros tipos, vea los artículos Operadores de igualdad y Operadores de comparación.

Prioridad de operadores

En la lista siguiente se ordenan los operadores relacionados con el puntero desde la prioridad más alta a la más baja:

- Operadores de incremento `x++` y decremento `x--` postfijos y operadores `->` y `[]`
- Operadores de incremento `++x` y decremento `--x` prefijos y operadores `&` y `*`
- Operadores `+` y `-` de suma
- Operadores de comparación `<`, `>`, `<=` y `>=`
- Operadores de igualdad `==` y `!=`

Use paréntesis, `()`, para cambiar el orden de evaluación impuesto por la prioridad de los operadores.

Para obtener la lista completa de los operadores de C# ordenados por nivel de prioridad, vea la sección Prioridad de operadores del artículo Operadores de C#.

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario no puede sobrecargar los operadores relacionados con el puntero `&`, `*`, `->` y `[]`.

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Variables fijas y móviles

- El operador address-of
- Direccionamiento indirecto del puntero
- Acceso a miembros de puntero
- Acceso de elemento de puntero
- Aritmética de puntero
- Incremento y decremento de puntero
- Comparación de punteros

Vea también

- Referencia de C#
- Operadores de C#
- Tipos de puntero
- unsafe (palabra clave)
- fixed (palabra clave)
- stackalloc
- sizeof (operador)

Operadores de asignación (referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

El operador de asignación `=` asigna el valor de su operando de la derecha a una variable, una propiedad o un elemento de indizador proporcionado por el operando de la izquierda. El resultado de una expresión de asignación es el valor asignado al operando izquierdo. El tipo del operando de la derecha debe ser el mismo que el del operando de la izquierda o debe poder convertirse implícitamente en él.

El operador de asignación `=` es asociativo a la derecha, es decir, una expresión con el formato

```
a = b = c
```

se evalúa como

```
a = (b = c)
```

En el ejemplo siguiente se muestra el uso del operador de asignación con una variable local, una propiedad y un elemento indexador como su operando izquierdo:

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

Operador de asignación ref

A partir C# 7.3, puede usar el operador de asignación ref `= ref` para reasignar una variable local de tipo ref o local de tipo ref readonly. En el siguiente ejemplo se muestra el uso del operador de asignación ref:

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

En el caso del operador de asignación de referencias, sus dos operandos deben ser del mismo tipo.

Asignación compuesta

Para un operador binario `op`, una expresión de asignación compuesta con el formato

```
x op= y
```

es equivalente a

```
x = x op y
```

salvo que `x` solo se evalúa una vez.

La asignación compuesta es compatible con operadores aritméticos, lógicos booleanos y de desplazamiento y lógicos bit a bit.

Asignación de uso combinado de NULL

A partir de C# 8.0, puede usar el operador de asignación de uso combinado de NULL `??=` para asignar el valor de su operando derecho al operando izquierdo solo si el operando izquierdo se evalúa como `null`. Para obtener más información, consulte Operadores ?? y ??.

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario no puede sobrecargar el operador de asignación. Sin embargo, un tipo definido por el usuario puede definir una conversión implícita a otro tipo. De este modo, el valor de un tipo definido por el usuario puede asignarse a una variable, una propiedad o un elemento de indizador de otro tipo. Para obtener más información, vea Operadores de conversión definidos por el usuario.

Un tipo definido por el usuario no puede sobrecargar de forma explícita un operador de asignación compuesta. Pero si un tipo definido por el usuario sobrecarga un operador binario `op`, el operador `op=`, si existe, también se sobrecarga de forma implícita.

especificación del lenguaje C#

Para más información, consulte la sección sobre operadores de asignación de la Especificación del lenguaje C#.

Para más información sobre el operador de asignación de referencias `= ref`, vea la nota de propuesta de

características.

Vea también

- Referencia de C#
- Operadores de C#
- ref (palabra clave)

Operadores + y += (referencia de C#)

04/05/2020 • 3 minutes to read • Edit Online

Los operadores `+` y `+=` son compatibles con los tipos numéricos enteros y de punto flotante, el tipo string y los tipos delegados.

Para obtener información acerca del operador aritmético `+`, consulte las secciones correspondientes a los operadores unarios más y menos y al operador de suma + del artículo Operadores aritméticos.

Concatenación de cadenas

Cuando uno o ambos operandos son de tipo string, el operador `+` concatena las representaciones de cadena de sus operandos (la representación de cadena de `null` es una cadena vacía):

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

A partir de C# 6, la interpolación de cadenas proporciona una manera más conveniente de dar formato a las cadenas:

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

Combinación de delegados

Para los operandos del mismo tipo de delegado, el operador `+` devuelve una nueva instancia de delegado que, cuando se invoca, invoca el operando de la izquierda y luego invoca el operando de la derecha. Si alguno de los operandos es `null`, el operador `+` devuelve el valor del otro operando (que también podría ser `null`). El ejemplo siguiente muestra cómo los delegados se pueden combinar con el operador `+`:

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab();  // output: ab
```

Para llevar a cabo la eliminación de delegados, utilice el operador `-`.

Para más información sobre los tipos de delegado, vea Delegados.

Operador de asignación y suma +=

Una expresión que usa el operador `+=`, como

```
x += y
```

es equivalente a

```
x = x + y
```

salvo que `x` solo se evalúa una vez.

En el siguiente ejemplo se muestra el uso del operador `+=`:

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer();  // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer();  // output: ab
```

También usa el operador `+=` para especificar un método de controlador de eventos cuando se suscribe a un evento. Para obtener más información, vea Procedimientos para suscribir y cancelar la suscripción a eventos.

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario puede sobrecargar el operador `+`. Cuando se sobrecarga un operador `+` binario, el operador `+=` también se sobrecarga de modo implícito. Un tipo definido por el usuario no puede sobrecargar de forma explícita el operador `+=`.

Especificación del lenguaje C#

Para más información, consulte las secciones Operador unario más y Operador de suma de la especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#
- Concatenación de varias cadenas
- Eventos
- Operadores aritméticos
- Operadores - y -=

Operadores - y -= (referencia de C#)

18/03/2020 • 5 minutes to read • Edit Online

Los operadores `-` y `-=` son compatibles con los tipos numéricos enteros y de punto flotante, y los tipos delegados.

Para obtener información sobre el operador aritmético `-`, consulte las secciones correspondientes a los operadores unarios más y menos y al operador de resta (-) del artículo Operadores aritméticos.

Eliminación de delegados

Para los operandos del mismo tipo delegado, el operador `-` devuelve una instancia de delegado que se calcula de la siguiente manera:

- Si ambos operandos no son nulos y la lista de invocación del operando derecho es una sublista apropiada contigua de la lista de invocación del operando izquierdo, el resultado de la operación es una nueva lista de invocación que se obtiene mediante la eliminación de las entradas del operando derecho de la lista de invocación del operando izquierdo. Si la lista del operando derecho coincide con varias sublistas contiguas en la lista del operando izquierdo, se quita solo la sublista coincidente más a la derecha. Si la eliminación da como resultado una lista vacía, el resultado es `null`.

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");

var abbaab = a + b + b + a + a + b;
abbaab();  // output: abbaab
Console.WriteLine();

var ab = a + b;
var abba = abbaab - ab;
abba();  // output: abba
Console.WriteLine();

var nihil = abbaab - abbaab;
Console.WriteLine(nihil is null);  // output: True
```

- Si la lista de invocación del operando derecho no es una sublista apropiada contigua de la lista de invocación del operando izquierdo, el resultado de la operación es el operando izquierdo. Por ejemplo, la eliminación de un delegado que no forma parte del delegado de multidifusión no surte ningún efecto y da como resultado que el delegado de multidifusión no cambie.

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");

var abbaab = a + b + b + a + a + b;
var aba = a + b + a;

var first = abbaab - aba;
first();  // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first));  // output: True

Action a2 = () => Console.Write("a");
var changed = aba - a;
changed();  // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged();  // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged));  // output: True
```

El ejemplo anterior también demuestra que, durante la eliminación de delegados, se comparan las instancias de delegados. Por ejemplo, los delegados que se producen de la evaluación de expresiones lambda idénticas no son iguales. Para obtener más información acerca de la igualdad de delegados, consulte la sección Operadores de igualdad de delegado de la especificación del lenguaje C#.

- Si el operando izquierdo es `null`, el resultado de la operación es `null`. Si el operando derecho es `null`, el resultado de la operación es el operando izquierdo.

```
Action a = () => Console.Write("a");

var nothing = null - a;
Console.WriteLine(nothing is null);  // output: True

var first = a - null;
a();  // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a));  // output: True
```

Para combinar delegados, utilice el operador `+`.

Para más información sobre los tipos de delegado, vea Delegados.

Operador de resta y asignación (-=)

Una expresión que usa el operador `-=`, como

```
x -= y
```

es equivalente a

```
x = x - y
```

salvo que `x` solo se evalúa una vez.

En el siguiente ejemplo se muestra el uso del operador `-=`:

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer();  // output: aba

Console.WriteLine();
printer -= a;
printer();  // output: ab
```

También se usa el operador `-=` con el fin de especificar un método de controlador de eventos para eliminar cuando se finaliza la suscripción a un evento. Para obtener más información, vea Procedimiento para suscribir y cancelar la suscripción a eventos.

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario puede sobrecargar el operador `-`. Cuando se sobrecarga un operador `-` binario, el operador `-=` también se sobrecarga de modo implícito. Un tipo definido por el usuario no puede sobrecargar de forma explícita el operador `-=`.

especificación del lenguaje C#

Para más información, consulte las secciones correspondientes al operador unario menos y al operador de resta de Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#
- Eventos
- Operadores aritméticos
- Operadores + y +=

Operador ?: (referencia de C#)

19/03/2020 • 4 minutes to read • Edit Online

El operador condicional `?:`, también conocido como operador condicional ternario, evalúa una expresión booleana y devuelve el resultado de una de las dos expresiones, en función de que la expresión booleana se evalúe como `true` o `false`.

La sintaxis del operador condicional es la siguiente:

```
condition ? consequent : alternative
```

La expresión `condition` debe evaluarse como `true` o `false`. Si `condition` se evalúa como `true`, se evalúa la expresión `consequent` y su resultado se convierte en el resultado de la operación. Si `condition` se evalúa como `false`, se evalúa la expresión `alternative` y su resultado se convierte en el resultado de la operación. Solo se evalúan `consequent` o `alternative`.

Los tipos de `consequent` y `alternative` deben coincidir o debe haber una conversión implícita de un tipo al otro.

El operador condicional es asociativo a la derecha, es decir, una expresión de la forma

```
a ? b : c ? d : e
```

se evalúa como

```
a ? b : (c ? d : e)
```

TIP

Puede utilizar el siguiente recurso mnemotécnico para recordar cómo se evalúa el operador condicional:

```
is this condition true ? yes : no
```

En el siguiente ejemplo se muestra el uso del operador condicional:

```
double sinc(double x) => x != 0.0 ? Math.Sin(x) / x : 1;

Console.WriteLine(sinc(0.1));
Console.WriteLine(sinc(0.0));
// Output:
// 0.998334166468282
// 1
```

Expresión condicional ref

A partir de C# 7.2, las variables locales de tipo ref o de tipo ref readonly se pueden asignar condicionalmente con la expresión condicional ref. La expresión condicional ref también se puede usar como un valor devuelto de referencia o como un método de argumento de `ref`.

La sintaxis de la expresión condicional ref es la siguiente:

```
condition ? ref consequent : ref alternative
```

Al igual que el operador condicional original, la expresión condicional ref evalúa solo una de las dos expresiones: ya sea `consequent` o `alternative`.

En el caso de la expresión condicional ref, los tipos de `consequent` y `alternative` deben coincidir.

En el siguiente ejemplo se muestra el uso de la expresión condicional ref:

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

Operador condicional y una instrucción `if..else`

Es posible que el uso del operador condicional en lugar de una instrucción if-else genere código más conciso en aquellos casos en los que tenga que calcular un valor condicionalmente. El ejemplo siguiente muestra dos maneras de clasificar un entero como negativo o no negativo:

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario no puede sobrecargar el operador condicional.

Especificación del lenguaje C#

Para más información, vea la sección sobre operadores condicionales de la Especificación del lenguaje C#.

Para más información sobre la expresión condicional ref, vea la nota de propuesta de características.

Vea también

- Referencia de C#
- Operadores de C#
- Instrucción if-else
- Operadores ?. y ?[]
- Operadores ?? y ??=
- ref (palabra clave)

! (permite valores NULL) (referencia de C#)

06/05/2020 • 5 minutes to read • Edit Online

Disponible en C# 8.0 y versiones posteriores, el operador `!` de postfijo unario es el operador que permite un valor NULL. En un contexto de anotación que admite un valor NULL habilitado, se usa el operador que permite un valor NULL para declarar que la expresión `x` de un tipo de referencia no es `null`: `x!`. El operador `!` de prefijo unario es el operador lógico de negación.

El operador que permite un valor NULL no tiene ningún efecto en tiempo de ejecución. Solo afecta al análisis de flujo estático del compilador al cambiar el estado NULL de la expresión. En tiempo de ejecución, la expresión `x!` se evalúa en el resultado de la expresión subyacente `x`.

> **NOTE**
>
> En C# 8, el operador null-forgiving finaliza la lista de operaciones null-conditional anteriores. Por ejemplo, la expresión `x?.y!.z` se analiza como `(x?.y)!.z`. Debido a esta interpretación, `z` se evalúa incluso si `x` es `null`, lo que puede dar lugar a NullReferenceException.

Para obtener más información sobre la característica de tipos de referencia que admiten un valor NULL, consulte Tipos de referencia que admiten un valor NULL.

Ejemplos

Uno de los casos de uso del operador que permite un valor NULL es probar la lógica de validación de argumentos. Por ejemplo, considere la siguiente clase:

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

Con el marco de pruebas MSTest puede crear la prueba siguiente para la lógica de validación en el constructor:

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Sin el operador que permite un valor NULL, el compilador genera la advertencia siguiente para el código anterior: `Warning CS8625: Cannot convert null literal to non-nullable reference type`. Al usar el operador que permite un valor NULL, se notifica al compilador que lo esperado es que se pase `null` y no se debe advertir al respecto.

También puede usar el operador que permite valores NULL cuando sepa a ciencia cierta que una expresión no puede ser `null`, pero el compilador no consigue reconocerlo. En el ejemplo siguiente, si el método `IsValid` devuelve `true`, su argumento no es `null` y puede desreferenciarlo de forma segura:

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
{
    return person != null && !string.IsNullOrEmpty(person.Name);
}
```

Sin el operador que permite un valor NULL, el compilador genera la advertencia siguiente para el código `p.Name`: `Warning CS8602: Dereference of a possibly null reference`.

Si puede modificar el método `IsValid`, puede usar el atributo NotNullWhen para notificar al compilador que un argumento del método `IsValid` no puede ser `null` cuando el método devuelve `true`:

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
{
    return person != null && !string.IsNullOrEmpty(person.Name);
}
```

En el ejemplo anterior, no es necesario usar el operador que permite un valor NULL porque el compilador tiene suficiente información para averiguar que `p` no puede ser `null` en la instrucción `if`. Para más información sobre los atributos que permiten proporcionar información adicional sobre el estado NULL de una variable, vea Actualización de las API con atributos para definir las expectativas NULL.

Especificación del lenguaje C#

Para obtener más información, consulte la sección The null-forgiving operator (El operador que admite valores NULL) del borrador de la especificación de tipos de referencia que aceptan valores NULL.

Vea también

- Referencia de C#
- Operadores de C#
- Tutorial: Diseño con tipos de referencia que admiten un valor NULL

?? Operadores ?? y ?? (referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

El operador de uso combinado de NULL `??` devuelve el valor del operando izquierdo si no es `null`; en caso contrario, evalúa el operando derecho y devuelve su resultado. El operador `??` no evalúa su operando derecho si el operando izquierdo se evalúa como no NULL.

Disponible en C# 8.0 y versiones posteriores, el operador de asignación de uso combinado de NULL `??=` asigna el valor de su operando derecho al operando izquierdo solo si el operando izquierdo se evalúa como `null`. El operador `??=` no evalúa su operando derecho si el operando izquierdo se evalúa como no NULL.

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers));  // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers));  // output: 5 0
Console.WriteLine(a);  // output: 0
```

El operando izquierdo del operador `??=` debe ser una variable, una propiedad o un elemento de indizador.

En C# 7.3 y versiones anteriores, el tipo del operando izquierdo del operador `??` debe ser un tipo de referencia o un tipo de valor que acepta valores NULL. A partir C# 8.0, ese requisito se reemplaza por lo siguiente: el tipo del operando izquierdo de los operadores `??` y `??=` no puede ser un tipo de valor que no acepta valores NULL. En concreto, a partir de C# 8.0 puede usar los operadores de fusión de NULL con parámetros de tipo sin restricciones:

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

Los operadores de uso combinado de NULL son asociativos a la derecha. Es decir, las expresiones del formulario

```
a ?? b ?? c
d ??= e ??= f
```

se evalúan como

```
a ?? (b ?? c)
d ??= (e ??= f)
```

Ejemplos

Los operadores `??` y `??=` puede resultar útiles en los siguientes escenarios:

- En expresiones con los operadores no condicionales ? y ?[], puede usar el operador `??` para proporcionar una expresión alternativa para evaluar en caso de que el resultado de la expresión con la

operación condicional NULL sea `null`:

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum = SumNumbers(null, 0);
Console.WriteLine(sum);  // output: NaN
```

- Cuando trabaja con tipos de valor que aceptan valores NULL y necesita proporcionar un valor de un tipo de valor subyacente, use el operador `??` para especificar el valor para proporcionar en caso de que un valor de tipo que acepta valores NULL sea `null`:

  ```
  int? a = null;
  int b = a ?? -1;
  Console.WriteLine(b);  // output: -1
  ```

  Use el método Nullable<T>.GetValueOrDefault() si el valor que se va usar cuando un valor de tipo que acepta valores NULL es `null` debe ser el valor predeterminado del tipo de valor subyacente.

- A partir de C# 7.0, puede usar una expresión `throw` como el operando derecho del operador `??` para hacer el código de comprobación de argumentos más conciso:

  ```
  public string Name
  {
      get => name;
      set => name = value ?? throw new ArgumentNullException(nameof(value), "Name cannot be null");
  }
  ```

  El ejemplo anterior también muestra cómo usar miembros con forma de expresión para definir una propiedad.

- A partir de C# 8.0, puede usar el operador `??=` para reemplazar el código del formulario

  ```
  if (variable is null)
  {
      variable = expression;
  }
  ```

  por el siguiente código:

  ```
  variable ??= expression;
  ```

Posibilidad de sobrecarga del operador

Los operadores `??` y `??=` no se pueden sobrecargar.

especificación del lenguaje C#

Para obtener más información sobre el operador `??`, vea la sección El operador de uso combinado de NULL de la especificación del lenguaje C#.

Para más información sobre el operador `??=`, consulte la nota de propuesta de características.

Vea también

- Referencia de C#
- Operadores de C#
- Operadores ?. y ?[]
- Operador ?:

Operador => (referencia de C#)

18/03/2020 • 3 minutes to read • Edit Online

El token `=>` se admite de dos formas: como el operador lambda y como un separador de un nombre de miembro y la implementación del miembro en una definición de cuerpo de expresión.

Operador{1}{2}lambda

En las expresiones lambda, el operador `=>` {4}lambda {5} separa los parámetros de entrada del lado izquierdo y el cuerpo lambda del lado derecho.

En el ejemplo siguiente se usa la característica LINQ con sintaxis de método para demostrar el uso de las expresiones lambda:

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

Los parámetros de entrada de una expresión lambda están fuertemente tipados en tiempo de compilación. Cuando el compilador puede deducir los tipos de los parámetros de entrada, como en el ejemplo anterior, se pueden omitir las declaraciones de tipos. Si tiene que especificar el tipo de los parámetros de entrada, debe hacerlo para cada uno de ellos, como se muestra en el ejemplo siguiente:

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product);   // output: 280
```

En el ejemplo siguiente se muestra cómo definir una expresión lambda sin parámetros de entrada:

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

Para obtener más información, vea Expresiones lambda.

Definición de cuerpo de expresiones

Una definición de cuerpo de expresión tiene la siguiente sintaxis general:

```
member => expression;
```

donde `expression` es una expresión válida. El tipo de valor devuelto de `expression` se debe poder convertir implícitamente al tipo de valor devuelto del miembro. Si el tipo de valor devuelto del miembro es `void`, o si el miembro es un constructor, un finalizador o un descriptor de acceso `set` de propiedad, `expression` debe ser una *expresión de instrucción*, que puede ser de cualquier tipo.

En el ejemplo siguiente se muestra una definición de cuerpo de expresión para un método `Person.ToString`:

```
public override string ToString() => $"{fname} {lname}".Trim();
```

Es una versión abreviada de la definición de método siguiente:

```
public override string ToString()
{
   return $"{fname} {lname}".Trim();
}
```

A partir de C# 6, se admiten definiciones de cuerpos de expresión para métodos, operadores y propiedades de solo lectura. A partir de C# 7.0, se admiten definiciones de cuerpos de expresión para constructores, finalizadores y descriptores de acceso de propiedad e indizador.

Para obtener más información, vea Miembros con forma de expresión.

Posibilidad de sobrecarga del operador

El operador `=>` no se puede sobrecargar.

especificación del lenguaje C#

Para más información sobre el operador lambda, vea la sección Expresiones de función anónima de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#

Operador :: (referencia de C#)

02/04/2020 • 2 minutes to read • Edit Online

Use el calificador de alias de espacio de nombres `::` para acceder a un miembro del espacio de nombres con alias. Solo puede usar el calificador `::` entre dos identificadores. El identificador de la izquierda puede ser cualquiera de los siguientes alias:

- Un alias de espacio de nombres creado con una directiva de alias using:

```
using forwinforms = System.Drawing;
using forwpf = System.Windows;

public class Converters
{
    public static forwpf::Point Convert(forwinforms::Point point) => new forwpf::Point(point.X,
point.Y);
}
```

- Un alias externo.

- El alias `global`, que es el alias del espacio de nombres global. El espacio de nombres global es el espacio de nombres que contiene los espacios de nombres y los tipos que no se declaran dentro de un espacio de nombres con nombre. Cuando se usa con el calificador `::`, el alias `global` siempre hace referencia al espacio de nombres global, incluso si está el alias del espacio de nombres `global` definido por el usuario.

  En el ejemplo siguiente se usa el alias `global` para tener acceso al espacio de nombres System de .NET, que es miembro del espacio de nombres global. Sin el alias `global`, se tendría acceso al espacio de nombres `System` definido por el usuario, que es miembro del espacio de nombres `MyCompany.MyProduct`:

```
namespace MyCompany.MyProduct.System
{
    class Program
    {
        static void Main() => global::System.Console.WriteLine("Using global alias");
    }

    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

> **NOTE**
>
> La palabra clave `global` es el alias de espacio de nombres global solo cuando es el identificador izquierdo del calificador `::`.

También puede usar el token `.` para acceder a un miembro de un espacio de nombres con alias. Sin embargo, el operador `.` también se usa para acceder a un miembro del tipo. El calificador `::` garantiza que el identificador de la izquierda siempre hace referencia a un alias de espacio de nombres, aunque exista un tipo o un espacio de nombres con el mismo nombre.

Especificación del lenguaje C#

Para más información, consulte la sección sobre calificadores de alias de espacio de nombres de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#
- Uso de espacios de nombres

Operador await (referencia de C#)

04/05/2020 • 6 minutes to read • Edit Online

El operador `await` suspende la evaluación del método async envolvente hasta que se completa la operación asincrónica representada por su operando. Cuando se completa la operación asincrónica, el operador `await` devuelve el resultado de la operación, si existe. Cuando el operador `await` se aplica al operando que representa una operación ya completada, devuelve el resultado de la operación inmediatamente sin suspender el método envolvente. El operador `await` no bloquea el subproceso que evalúa el método async. Cuando el operador `await` suspende el método async envolvente, el control vuelve al autor de la llamada del método.

En el ejemplo siguiente, el método HttpClient.GetByteArrayAsync devuelve la instancia `Task<byte[]>`, que representa una operación asincrónica que genera una matriz de bytes cuando se completa. Hasta que se complete la operación, el operador `await` suspende el método `DownloadDocsMainPageAsync`. Cuando se suspende `DownloadDocsMainPageAsync`, se devuelve el control `Main` al método, que es el autor de la llamada a `DownloadDocsMainPageAsync`. El método `Main` se ejecuta hasta que necesita el resultado de la operación asincrónica realizada por el método `DownloadDocsMainPageAsync`. Cuando GetByteArrayAsync obtiene todos los bytes, se evalúa el resto del método `DownloadDocsMainPageAsync`. Después, se evalúa el resto del método `Main`.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocsMainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded} bytes.");
    }

    private static async Task<int> DownloadDocsMainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: About to start downloading.");

        var client = new HttpClient();
        byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: Finished downloading.");
        return content.Length;
    }
}
// Output similar to:
// DownloadDocsMainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocsMainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

En el ejemplo anterior se usa el método async `Main`, lo que es posible a partir de C# 7.1. Para obtener más información, vea la sección Operador await en el método Main.

> **NOTE**
> Para obtener una introducción a la programación asincrónica, vea Programación asincrónica con async y await. La programación asincrónica con `async` y `await` sigue el modelo asincrónico basado en tareas.

El operador `await` se puede usar solamente en un método, una expresión lambda o un método anónimo modificado por la palabra clave async. Dentro de un método async, no se puede usar el operador `await` en el cuerpo de una función sincrónica, dentro del bloque de una instrucción lock y en un contexto no seguro.

El operando del operador `await` suele ser de uno de los siguientes tipos de .NET: Task, Task<TResult>, ValueTask o ValueTask<TResult>. Aunque cualquier expresión con await puede ser el operando del operador `await`. Para obtener más información, vea la sección Expresiones con await de la especificación del lenguaje C#.

A partir C# 8.0, puede usar la instrucción `await foreach` para utilizar un flujo de datos asincrónico. Para obtener más información, vea el artículo sobre la instrucción `foreach` y la sección Secuencias asincrónicas del artículo Novedades de C# 8.0.

El tipo de expresión `await t` es `TResult` si el tipo de expresión `t` es Task<TResult> o ValueTask<TResult>. Si el tipo de `t` es Task o ValueTask, el tipo de `await t` es `void`. En ambos casos, si `t` produce una excepción, `await t` vuelve a producir la excepción. Para obtener más información sobre cómo controlar las excepciones, vea la sección Excepciones en métodos async del artículo Instrucción try-catch.

Las palabras clave `async` y `await` están disponibles en C# 5 y versiones posteriores.

Operador await en el método Main

A partir C# 7.1, el método `Main`, que es el punto de entrada de la aplicación, puede devolver `Task` o `Task<int>`, lo que le permite ser asincrónico para poder usar el operador `await` en su cuerpo. En versiones anteriores de C#, para tener la certeza de que el método `Main` espera a que finalice una operación asincrónica, puede recuperar el valor de la propiedad Task<TResult>.Result de la instancia Task<TResult> devuelta por el método async correspondiente. Para las operaciones asincrónicas que no generan un valor, puede llamar al método Task.Wait. Para obtener información sobre cómo seleccionar la versión del lenguaje, vea Control de versiones del lenguaje C#.

Especificación del lenguaje C#

Para obtener más información, vea la sección Expresiones await de la especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#
- async
- Modelo de programación asincrónica de tareas
- Programación asincrónica
- Async en profundidad
- Tutorial: Acceso a la web con async y await
- Tutorial: Generación y uso de secuencias asincrónicas con C# 8.0 y .NET Core 3.0

Expresiones de valor predeterminado (referencia de C#)

02/04/2020 • 2 minutes to read • Edit Online

Una expresión de valor predeterminado genera el valor predeterminado de un tipo. Hay dos tipos de expresiones de valor predeterminado: la llamada al operador predeterminado y un literal predeterminado.

También se usa la palabra clave `default` como etiqueta de mayúsculas y minúsculas predeterminada dentro de una instrucción `switch`.

operador default

El argumento del operador `default` debe ser el nombre de un tipo o un parámetro de tipo, como se muestra en el ejemplo siguiente:

```
Console.WriteLine(default(int));  // output: 0
Console.WriteLine(default(object) is null);  // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ? "null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

Literal default

A partir C# de 7.1, puede usar el literal `default` para generar el valor predeterminado de un tipo cuando el compilador puede deducir el tipo de expresión. La expresión literal `default` genera el mismo valor que la expresión `default(T)` cuando se deduce el tipo `T`. Puede usar el literal `default` en cualquiera de los casos siguientes:

- En la asignación o inicialización de una variable.
- En la declaración del valor predeterminado de un parámetro de método opcional.
- En una llamada al método para proporcionar un valor de argumento.
- En una instrucción `return` o como una expresión de un miembro con cuerpo de expresión.

En el ejemplo siguiente se muestra el uso del literal `default`:

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        return default;
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ", values)} ]");

Display(InitializeArray<int>(3));  // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default));  // output: [ False, False, False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue));  // output: [ (0, 0), (0, 0), (0, 0) ]
```

Especificación del lenguaje C#

Para más información, consulte la sección Expresiones de valor predeterminado de la especificación del lenguaje C#.

Para más información sobre el literal `default`, consulte la nota de propuesta de características.

Vea también

- Referencia de C#
- Operadores de C#
- Valores predeterminados de los tipos de C#
- Elementos genéricos en .NET

Operador delegate (referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

El operador `delegate` crea un método anónimo que se puede convertir en un tipo delegado:

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4));  // output: 7
```

NOTE

A partir de C# 3, las expresiones lambda proporcionan una manera más concisa y expresiva de crear una función anónima. Use el operador => para construir una expresión lambda:

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4));  // output: 7
```

Para más información sobre las características de las expresiones lambda, como capturar las variables externas, consulte Expresiones lambda.

Al usar el operador `delegate`, puede omitir la lista de parámetros. Si lo hace, el método anónimo creado se puede convertir en un tipo delegado con cualquier lista de parámetros, tal como se muestra en el ejemplo siguiente:

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

Esta es la única funcionalidad de los métodos anónimos que las expresiones lambda no admiten. En todos los demás casos, una expresión lambda es una de las maneras preferidas de escribir código alineado.

También puede usar la palabra clave `delegate` para declarar un tipo delegado.

especificación del lenguaje C#

Para obtener más información, vea la sección Expresiones de función anónima de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#
- Operador =>

Expresión nameof (referencia de C#)

04/05/2020 • 2 minutes to read • Edit Online

Una expresión `nameof` genera el nombre de una variable, un tipo o un miembro como constante de cadena:

```
Console.WriteLine(nameof(System.Collections.Generic));  // output: Generic
Console.WriteLine(nameof(List<int>));  // output: List
Console.WriteLine(nameof(List<int>.Count));  // output: Count
Console.WriteLine(nameof(List<int>.Add));  // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers));  // output: numbers
Console.WriteLine(nameof(numbers.Count));  // output: Count
Console.WriteLine(nameof(numbers.Add));  // output: Add
```

Como se muestra en el ejemplo anterior, en el caso de un tipo y un espacio de nombres, el nombre generado por lo general no está completo.

En el caso de identificadores textuales, el carácter `@` no es parte de un nombre, como se muestra en el ejemplo siguiente:

```
var @new = 5;
Console.WriteLine(nameof(@new));  // output: new
```

La expresión `nameof` se evalúa en tiempo de compilación y no tiene efecto en tiempo de ejecución.

Puede usar una expresión `nameof` para facilitar el mantenimiento del código de comprobación de argumentos:

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"{nameof(Name)} cannot be null");
}
```

Una expresión `nameof` está disponible en C# 6 y versiones posteriores.

Especificación del lenguaje C#

Para más información, consulte la sección Expresiones Nameof de la especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#

Operador new (referencia de C#)

06/05/2020 • 3 minutes to read • Edit Online

El operador `new` crea una nueva instancia de un tipo.

También puede usar la palabra clave `new` como un modificador de declaración de miembro o una restricción de tipo genérico.

Invocación del constructor

Para crear una nueva instancia de un tipo, normalmente se invoca uno de los constructores de ese tipo mediante el operador `new`:

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Puede usar un inicializador de colección u objeto con el operador `new` para crear una instancia e inicializar un objeto en una sola instrucción, como se muestra en el ejemplo siguiente:

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

creación de matriz

También se usa el operador `new` para crear una instancia de matriz, como se muestra en el ejemplo siguiente:

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

Utilice la sintaxis de inicialización de matriz para crear una instancia de matriz y rellenarla con los elementos en una sola instrucción. En el ejemplo siguiente se muestran varias maneras de hacerlo:

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType());  // output: System.Int32[]
```

Para obtener más información sobre las matrices, consulte Matrices.

Creación de instancias de tipos anónimos

Para crear una instancia de un tipo anónimo, utilice el operador `new` y la sintaxis de inicializador de objeto:

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

Destrucción de instancias de tipo

No tiene que destruir instancias de tipo creadas anteriormente. Las instancias de tipos de valor y referencia se destruyen automáticamente. Las instancias de tipos de valor se destruyen en cuanto se destruye el contexto que los contiene. Las instancias de tipos de referencia los destruye el recolector de elementos no utilizados en un momento no especificado después de eliminarse la última referencia a ellos.

Para los tipos de instancia que contienen recursos no administrados, como un identificador de archivo, se recomienda realizar una limpieza determinista para asegurarse de que los recursos que contienen se liberan tan pronto como sea posible. Para más información, vea la referencia de la API System.IDisposable y el artículo sobre la instrucción using.

Posibilidad de sobrecarga del operador

Un tipo definido por el usuario no puede sobrecargar el operador `new`.

Especificación del lenguaje C#

Para más información, vea la sección El operador new de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#
- Inicializadores de objeto y colección

Operador sizeof (referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

El operador `sizeof` devuelve el número de bytes ocupados por una variable de un tipo determinado. El argumento del operador `sizeof` debe ser el nombre de un tipo administrado o un parámetro de tipo que está restringido para ser un tipo no administrado.

El operador `sizeof` requiere un contexto de unsafe. Sin embargo, las expresiones presentadas en la tabla siguiente se evalúan en tiempo de compilación según los valores de constantes correspondientes y no necesitan un contexto de unsafe:

EXPRESIÓN	VALOR CONSTANTE
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(decimal)`	16
`sizeof(bool)`	1

Tampoco es necesario usar un contexto de unsafe cuando el operando del operador `sizeof` es el nombre de un tipo enum.

En el siguiente ejemplo se muestra el uso del operador `sizeof`:

```
using System;

public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte));  // output: 1
        Console.WriteLine(sizeof(double));  // output: 8

        DisplaySizeOf<Point>();  // output: Size of Point is 24
        DisplaySizeOf<decimal>();  // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*));  // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}
```

El operador `sizeof` devuelve un número de bytes que asignará Common Language Runtime en la memoria administrada. Para los tipos struct, el valor incluye el relleno, tal y como se muestra en el ejemplo anterior. El resultado del operador `sizeof` puede ser distinto del resultado del método Marshal.SizeOf, que devuelve el tamaño de un tipo en la memoria *no administrada*.

especificación del lenguaje C#

Para obtener más información, vea la sección Operador sizeof de la Especificación del lenguaje C#.

Vea también

- Referencia de C#
- Operadores de C#
- Operadores relacionados con el puntero
- Tipos de puntero
- Tipos relacionados con el intervalo y la memoria
- Elementos genéricos en .NET

Expresión stackalloc (referencia de C#)

02/04/2020 • 5 minutes to read • Edit Online

La expresión `stackalloc` asigna un bloque de memoria en la pila. Un bloque de memoria asignado a la pila creado durante la ejecución del método se descarta automáticamente cuando se devuelva dicho método. No puede liberar explícitamente memoria asignada con `stackalloc`. Un bloque de memoria asignada a la pila no está sujeto a la recolección de elementos no utilizados y no tiene que fijarse con una instrucción `fixed`.

Puede asignar el resultado de una expresión `stackalloc` a una variable de uno de los siguientes tipos:

- A partir de C# 7.2, System.Span<T> o System.ReadOnlySpan<T>, como se muestra en el ejemplo siguiente:

  ```
  int length = 3;
  Span<int> numbers = stackalloc int[length];
  for (var i = 0; i < length; i++)
  {
      numbers[i] = i;
  }
  ```

  No tiene que usar un contexto unsafe al asignar un bloque de memoria asignado a la pila para una variable Span<T> o ReadOnlySpan<T>.

  Cuando se trabaja con esos tipos, puede usar una expresión `stackalloc` en expresiones condicionales o de asignación, como se muestra en el ejemplo siguiente:

  ```
  int length = 1000;
  Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new byte[length];
  ```

  A partir de C# 8.0, se puede usar una expresión `stackalloc` dentro de otras expresiones siempre que se permita una variable Span<T> o ReadOnlySpan<T>, como se muestra en el ejemplo siguiente:

  ```
  Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
  var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6 ,8 });
  Console.WriteLine(ind);  // output: 1
  ```

  > **NOTE**
  >
  > Se recomienda usar los tipos Span<T> o ReadOnlySpan<T> para trabajar con memoria asignada a la pila siempre que sea posible.

- Un tipo de puntero, como se muestra en el ejemplo siguiente:

```
unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

Como se muestra en el ejemplo anterior, se debe utilizar un contexto `unsafe` cuando se trabaja con tipos de puntero.

En el caso de los tipos de puntero, solo se puede usar una expresión `stackalloc` en una declaración de variable local para inicializar la variable.

La cantidad de memoria disponible en la pila es limitada. Si asigna demasiada memoria en la pila, se produce una excepción StackOverflowException. Para evitarlo, siga estas reglas:

- Limite la cantidad de memoria asignada con `stackalloc`:

  ```
  const int MaxStackLimit = 1024;
  Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc byte[inputLength] : new
  byte[inputLength];
  ```

  Como la cantidad de memoria disponible en la pila depende del entorno en el que se ejecuta el código, debe ser conservador al definir el valor límite real.

- Evite el uso de `stackalloc` dentro de bucles. Asigne el bloque de memoria fuera de un bucle y vuelva a usarlo dentro del bucle.

El contenido de la memoria recién asignada está sin definir. Debe inicializarlo antes del uso. Por ejemplo, puede usar el método Span<T>.Clear que establece todos los elementos en el valor predeterminado de tipo `T`.

A partir de C# 7.3, puede usar la sintaxis de inicializador de matriz para definir el contenido de la memoria recién asignada. En el ejemplo siguiente se muestran diversas formas de hacerlo:

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };
```

En la expresión `stackalloc T[E]`, `T` debe ser un tipo no administrado y `E` debe evaluarse como un valor int no negativo.

Seguridad

El uso de `stackalloc` habilita automáticamente las características de detección de saturación del búfer en el entorno Common Language Runtime (CLR). Si se detecta saturación del búfer, se finaliza el proceso lo antes posible para minimizar el riesgo de que se ejecute código malintencionado.

Especificación del lenguaje C#

Para más información, consulte la sección sobre asignación de pila de la especificación del lenguaje C# y la nota de la característica Permitir `stackalloc` en contextos anidados.

Vea también

- Referencia de C#
- Operadores de C#
- Operadores relacionados con el puntero
- Tipos de puntero
- Tipos relacionados con el intervalo y la memoria
- Qué hacer y qué no hacer de stackalloc

Expresión switch (referencia de C#)

04/05/2020 • 6 minutes to read • Edit Online

En este artículo se trata la expresión `switch`, introducida en C# 8.0. Para obtener información sobre la instrucción `switch`, consulte el artículo sobre la instrucción `switch` en la sección Instrucciones.

Ejemplo básico

La expresión `switch` se proporciona para una semántica similar a `switch` en un contexto de expresión. Proporciona una sintaxis concisa cuando los segmentos modificadores producen un valor. En el ejemplo siguiente se muestra la estructura de una expresión switch. Traslada los valores desde una `enum` que representa las direcciones visuales de un mapa en línea hasta la dirección cardinal correspondiente:

```
public static class SwitchExample
{
    public enum Directions
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static void Main()
    {
        var direction = Directions.Right;
        Console.WriteLine($"Map view direction is {direction}");

        var orientation = direction switch
        {
            Directions.Up    => Orientation.North,
            Directions.Right => Orientation.East,
            Directions.Down  => Orientation.South,
            Directions.Left  => Orientation.West,
        };
        Console.WriteLine($"Cardinal orientation is {orientation}");
    }
}
```

En el ejemplo anterior se muestran los elementos básicos de una expresión switch:

- La *expresión de intervalo*: En el ejemplo anterior se usa la variable `direction` como expresión de intervalo.
- Los *segmentos de expresión switch*: Cada segmento de expresión switch contiene un *patrón*, una *restricción de caso*, el token `=>` y una *expressión*.

El resultado de la *expresión switch* es el valor de la expresión del primer *segmento de expresión switch* cuyo *patrón* coincide con la *expresión de intervalo* y cuya *restricción de caso*, en caso de estar presente, se evalúa como `true`. La *expresión* situada a la derecha del token `=>` no puede ser una instrucción de expresión.

Los *segmentos de expresión switch* se evalúan en orden de texto. El compilador emite un error cuando no se puede elegir un *segmento de expresión switch* inferior porque un *segmento de expresión switch* superior coincide con todos sus valores.

Patrones y restricciones de caso

Muchos patrones se admiten en los segmentos de expresión switch. En el ejemplo anterior se usó en *patrón de valor*. Un *patrón de valor* compara la expresión de intervalo con un valor. Ese valor debe ser una constante de tiempo de compilación. El *patrón de valor* compara la expresión de intervalo con un tipo conocido. En el siguiente ejemplo se recupera el tercer elemento de una secuencia. Usa distintos métodos en función del tipo de la secuencia:

```
public static T TypeExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array array => (T)array.GetValue(2),
        IList<T> list      => list[2],
        IEnumerable<T> seq => seq.Skip(2).First(),
    };
```

Los patrones pueden ser recursivos, donde un patrón prueba un tipo, y si ese tipo coincide, el patrón coincide con uno o varios valores de propiedad en la expresión de intervalo. Puede usar patrones recursivos para ampliar el ejemplo anterior. Agregue segmentos de expresión switch para matrices que tengan menos de tres elementos. Los patrones recursivos se muestran en el ejemplo siguiente:

```
public static T RecursiveExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}       => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array               => (T)array.GetValue(2),
        IList<T> list                    => list[2],
        IEnumerable<T> seq               => seq.Skip(2).First(),
    };
```

Los patrones recursivos pueden examinar las propiedades de la expresión de intervalo, pero no pueden ejecutar código arbitrario. Puede usar una *restricción de caso*, especificada en una cláusula `when`, para proporcionar comprobaciones similares para otros tipos de secuencia:

```
public static T CaseGuardExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}                => default(T),
        System.Array { Length : 1} array          => (T)array.GetValue(0),
        System.Array { Length : 2} array          => (T)array.GetValue(1),
        System.Array array                        => (T)array.GetValue(2),
        IEnumerable<T> list when !list.Any()      => default(T),
        IEnumerable<T> list when list.Count() < 3 => list.Last(),
        IList<T> list                             => list[2],
        IEnumerable<T> seq                        => seq.Skip(2).First(),
    };
```

Por último, puede agregar los patrones `_` y `null` para detectar argumentos no procesados por ningún otro segmento de expresión switch. Eso hace que la expresión switch sea *exhaustiva*, lo que significa que se controla cualquier valor posible de la expresión de intervalo. En el siguiente ejemplo se agregan esos segmentos de expresión:

```
public static T ExhaustiveExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}       => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array               => (T)array.GetValue(2),
        IEnumerable<T> list
            when !list.Any()             => default(T),
        IEnumerable<T> list
            when list.Count() < 3        => list.Last(),
        IList<T> list                    => list[2],
        null                             => throw new ArgumentNullException(nameof(sequence)),
        _                                => sequence.Skip(2).First(),
    };
```

En el ejemplo anterior se agrega un patrón `null` y se cambia el patrón de tipo `IEnumerable<T>` a un patrón `_`. El patrón `null` proporciona una comprobación de NULL como segmento de expresión switch. La expresión para ese segmento inicia una excepción ArgumentNullException. El patrón `_` coincide con todas las entradas que no han coincidido con segmentos anteriores. Debe aparecer después de la comprobación `null` o coincidiría con las entradas `null`.

Puede leer más en la propuesta de especificación de lenguaje C# para patrones recursivos.

Vea también

- Referencia de C#
- Operadores de C#
- Coincidencia de patrones

Operadores true y false (referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

El operador `true` devuelve el valor bool `true` para indicar que su operando es definitivamente true. El operador `false` devuelve el valor `bool``true` para indicar que su operando es definitivamente false. Los operadores `true` y `false` no garantizan que se complementan entre sí. Es decir, tanto el operador `true` como `false` podrían devolver el valor `bool``false` del mismo operando. Si un tipo define uno de los dos operadores, también debe definir otro operador.

> **TIP**
>
> Use el tipo `bool?`, si tiene que admitir la lógica de tres valores (por ejemplo, cuando trabaja con bases de datos que admiten un tipo booleano de tres valores). C# proporciona los operadores `&` y `|` que admiten la lógica de tres valores con los operandos `bool?`. Para más información, consulte la sección Operadores lógicos booleanos que aceptan valores NULL del artículo Operadores lógicos booleanos.

Expresiones booleanas

Un tipo con el operador `true` definido puede ser el tipo de un resultado de una expresión condicional de control en las instruciones if, do, while y for y en el operador condicional `?:`. Para más información, vea la sección Expresiones booleanas de la Especificación del lenguaje C#.

Operadores lógicos condicionales definidos por el usuario

Si un tipo con los operadores `true` y `false` definidos sobrecarga el operador lógico OR `|` o el operador lógico AND `&` de una manera determinada, el operador lógico condicional OR `||` o el operador lógico condicional AND `&&`, respectivamente, se puede evaluar para los operandos de ese tipo. Para obtener más información, vea la sección Operadores lógicos condicionales definidos por el usuario de la Especificación del lenguaje C#.

Ejemplo

El ejemplo siguiente muestra el tipo que define los operadores `true` y `false`. Además, el tipo sobrecarga el operador lógico AND `&` de manera que el operador `&&` también se puede evaluar para los operandos de ese tipo.

```
using System;

public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x == Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) => x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x == y);

    public override bool Equals(object obj) => obj is LaunchStatus other && this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() && GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}
```

Tenga en cuenta el comportamiento de cortocircuito del operador `&&`. Cuando el método `GetFuelLaunchStatus` devuelve `LaunchStatus.Red`, el operando derecho del operador `&&` no se evalúa. Eso es porque `LaunchStatus.Red` es definitivamente false. A continuación, el resultado de AND lógico no depende del valor del operando derecho. El resultado del ejemplo es el siguiente:

```
Getting fuel launch status...
Wait!
```

Vea también

- Referencia de C#
- Operadores de C#

Sobrecarga de operadores (referencia de C#)

29/06/2020 • 7 minutes to read • Edit Online

Un tipo definido por el usuario puede sobrecargar un operador de C# predefinido. Es decir, un tipo puede proporcionar la implementación personalizada de una operación cuando uno o los dos operandos son de ese tipo. En la sección Operadores sobrecargables se muestra qué operadores de C# pueden sobrecargarse.

Use la palabra clave `operator` para declarar un operador. Una declaración de operador debe cumplir las reglas siguientes:

- Incluye los modificadores `public` y `static`.
- Un operador unario tiene un parámetro de entrada. Un operador binario tiene dos parámetros de entrada. En cada caso, al menos un parámetro debe ser de tipo `T` o `T?` donde `T` es el tipo que contiene la declaración del operador.

En el ejemplo siguiente se muestra una estructura simplificada para representar un número racional. La estructura sobrecarga algunos de los operadores aritméticos:

```
using System;

public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.", nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a);   // output: -5 / 4
        Console.WriteLine(a + b);  // output: 14 / 8
        Console.WriteLine(a - b);  // output: 6 / 8
        Console.WriteLine(a * b);  // output: 5 / 8
        Console.WriteLine(a / b);  // output: 10 / 4
    }
}
```

Puede ampliar el ejemplo anterior mediante la definición de una conversión implícita de `int` a `Fraction`. A continuación, los operadores sobrecargados admiten argumentos de esos dos tipos. Es decir, sería posible agregar un valor entero a una fracción y obtener como resultado una fracción.

También usa la palabra clave `operator` para definir una conversión de tipos personalizada. Para obtener más información, vea Operadores de conversión definidos por el usuario.

Operadores sobrecargables

En la tabla siguiente se proporciona información sobre la posibilidad de sobrecarga de los operadores de C#:

OPERADORES	POSIBILIDAD DE SOBRECARGA
+x, -x, !x, ~x, ++, --, true, false	Estos operadores unarios se pueden sobrecargar.
x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x == y, x != y, x < y, x > y, x <= y, x >= y	Estos operadores binarios se pueden sobrecargar. Determinados operadores deben sobrecargarse en pares; para obtener más información, vea la nota que está a continuación de esta tabla.
x && y, x || y	No se pueden sobrecargar los operadores lógicos condicionales. Sin embargo, si un tipo con los operadores `true` y `false` sobrecargados, también sobrecarga el operador `&` o `|` de determinada manera, el operador `&&` o `||`, respectivamente, se puede evaluar para los operandos de ese tipo. Para obtener más información, vea la sección Operadores lógicos condicionales definidos por el usuario de la Especificación del lenguaje C#.
a[i], `a?[i]`	El acceso a un elemento no se considera un operador sobrecargable, pero puede definir un indizador.
(T)x	No se puede sobrecargar el operador de conversión, pero pueden definirse conversiones de tipos personalizadas que pueden realizarse mediante una expresión de conversión. Para obtener más información, vea Operadores de conversión definidos por el usuario.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Los operadores de asignación compuestos no pueden sobrecargarse explícitamente. Pero cuando se sobrecarga un operador binario, el operador de asignación compuesto correspondiente, si lo hay, también se sobrecarga de modo implícito. Por ejemplo, `+=` se evalúa con `+`, que se pueden sobrecargar.
^x, x = y, x.y, `x?.y`, c ? t : f, x ?? y, x ??= y, x..y, x->y, =>, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, typeof	Estos operadores no se pueden sobrecargar.

NOTE

Los operadores de comparación deben sobrecargarse en pares. Es decir, si se sobrecarga un operador de un par, el otro operador debe sobrecargarse también. Se emparejan de la siguiente manera:

- Operadores `==` y `!=`
- Operadores `<` y `>`
- Operadores `<=` y `>=`

Especificación del lenguaje C#

Para más información, vea las secciones siguientes de la Especificación del lenguaje C#:

- Sobrecarga de operadores
- Operadores

Vea también

- Referencia de C#
- Operadores de C#
- Operadores de conversión definidos por el usuario
- Directrices de diseño: sobrecargas de operador
- Directrices de diseño: operadores de igualdad
- Why are overloaded operators always static in C#? (¿Por qué los operadores sobrecargados son siempre estáticos en C#?)

Caracteres especiales de C#

18/03/2020 • 2 minutes to read • Edit Online

Los caracteres especiales son caracteres contextuales predefinidos que modifican el elemento de programa (una cadena literal, un identificador o un nombre de atributo) para que se antepongan. C# admite los siguientes caracteres especiales:

- @, el carácter de identificador textual.
- $, el carácter de cadena interpolada.

Vea también

- Referencia de C#
- Guía de programación de C#

$ - Interpolación de cadenas: referencia de C#

04/05/2020 • 8 minutes to read • Edit Online

El carácter especial `$` identifica un literal de cadena como una *cadena interpolada*. Una cadena interpolada es un literal de cadena que puede contener *expresiones de interpolación*. Cuando una cadena interpolada se resuelve en una cadena de resultado, los elementos con expresiones de interpolación se reemplazan por las representaciones de cadena de los resultados de la expresión. Esta característica está disponible a partir de C# 6.

La interpolación de cadenas proporciona una sintaxis más legible y cómoda de crear cadenas con formato que si se usa la característica de formato compuesto de cadena. En este ejemplo se usan ambas características para producir el mismo resultado:

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

Estructura de una cadena interpolada

Para distinguir un literal de cadena como una cadena interpolada, antepóngale el símbolo `$`. No puede haber ningún espacio en blanco entre el carácter `$` y el carácter `"` que inicia un literal de cadena.

La estructura de un elemento con una expresión de interpolación es como se muestra aquí:

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

Los elementos entre corchetes son opcionales. En esta tabla se describe cada elemento:

ELEMENTO	DESCRIPCIÓN
`interpolationExpression`	Expresión que genera un resultado al que se va a aplicar formato. La representación de cadena de `null` es String.Empty.
`alignment`	La expresión constante cuyo valor define el número mínimo de caracteres en la representación de cadena del resultado de la expresión. Si es positivo, la representación de cadena está alineada a la derecha; si es negativo, está alineada a la izquierda. Para más información, vea Alignment (Componente).
`formatString`	Cadena de formato compatible con el tipo de resultado de la expresión. Para más información, vea Format String (Componente).

En el ejemplo siguiente, se usan componentes opcionales de formato que se han descrito anteriormente:

```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi
number");
// Expected output is:
// |Left   |  Right|
//     3.14159265358979 - default formatting of the pi number
//                3.142 - display only three decimal digits of the pi number
```

Caracteres especiales

Para incluir una llave ("{" o "}") en el texto generado por una cadena interpolada, use dos llaves ("{{" o "}}"). Para más información, vea Llaves de escape.

Como los dos puntos (":") tienen un significado especial en un elemento de expresión de interpolación, para poder usar un operador condicional en una expresión de interpolación, incluya esa expresión entre paréntesis.

En este ejemplo, se muestra cómo incluir una llave en una cadena de resultado y cómo usar un operador condicional en una expresión de interpolación:

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

Las cadenas textuales interpoladas comienzan por el carácter `$`, seguido del carácter `@`. Para más información sobre las cadenas textuales, vea los temas string e Identificador textual.

> **NOTE**
> A partir de C# 8.0, puede usar los tokens `$` y `@` en cualquier orden; tanto `$@"..."` como `@$"..."` son cadenas textuales interpoladas válidas. En versiones de C# anteriores, el token `$` debe aparecer delante del token `@`.

Conversiones implícitas y cómo especificar la implementación de `IFormatProvider`

Hay tres conversiones implícitas de una cadena interpolada:

1. Conversión de una cadena interpolada a una instancia de String, que es el resultado de la resolución de cadenas interpoladas con elementos de expresión de interpolación que se reemplazan con las representaciones de cadena con formato correcto de sus resultados. Esta conversión utiliza CurrentCulture para dar formato a los resultados de la expresión.

2. Conversión de una cadena interpolada a una instancia FormattableString que representa una cadena de formato compuesto junto con los resultados de expresión a los que se va a aplicar formato. Esto permite crear varias cadenas de resultado con contenido específico de la referencia cultural de una sola instancia de FormattableString. Para ello, llame a uno de estos métodos:

- Una sobrecarga de ToString() que genera una cadena de resultado para CurrentCulture.
- Método Invariant que genera una cadena de resultado para InvariantCulture.
- Un método ToString(IFormatProvider) que genera una cadena de resultado para una referencia cultural especificada.

También puede usar el método ToString(IFormatProvider) para proporcionar una implementación definida por el usuario de la interfaz IFormatProvider que admite formatos personalizados. Para más información, consulte la sección Formato personalizado con ICustomFormatter del artículo Aplicar formato a tipos en .NET.

3. Conversión de una cadena interpolada a una instancia IFormattable que también permite crear varias cadenas de resultado con contenido específico de la referencia cultural de una sola instancia IFormattable.

En el ejemplo siguiente, se usa la conversión implícita a FormattableString para crear cadenas de resultado específicas de la referencia cultural:

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

System.Globalization.CultureInfo.CurrentCulture =
System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10}
{messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");
// Expected output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

Recursos adicionales

Si no está familiarizado con la interpolación de cadenas, vea el tutorial interactivo Interpolación de cadenas en C#. También puede consultar otro tutorial de interpolación de cadenas en C# que muestra cómo usar cadenas interpoladas para generar cadenas con formato.

Compilación de cadenas interpoladas

Si una cadena interpolada tiene el tipo `string`, normalmente se transforma en una llamada de método String.Format. El compilador puede reemplazar String.Format por String.Concat si el comportamiento analizado es equivalente a una concatenación.

Si una cadena interpolada tiene el tipo IFormattable o FormattableString, el compilador genera una llamada al método FormattableStringFactory.Create.

Especificación del lenguaje C#

Para obtener más información, vea la sección Interpolated strings (Cadenas interpoladas) de la especificación del lenguaje C#.

Vea también

- Referencia de C#
- Caracteres especiales de C#
- Cadenas
- Cadenas con formato numérico estándar
- Formatos compuestos
- String.Format

@ (Referencia de C#)

02/04/2020 • 4 minutes to read • Edit Online

El carácter especial `@` actúa como un identificador textual. Se puede usar de estas formas:

1. Para habilitar el uso de palabras clave de C# como identificadores. El carácter `@` actúa como prefijo de un elemento de código que el compilador debe interpretar como un identificador en lugar de como una palabra clave de C#. En el ejemplo siguiente se usa el carácter `@` para definir un identificador denominado `for` que se usa en un bucle `for`.

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
   Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//     Here is your gift, John!
//     Here is your gift, James!
//     Here is your gift, Joan!
//     Here is your gift, Jamie!
```

2. Para indicar que un literal de cadena se debe interpretar literalmente. El carácter `@` de esta instancia define un *literal de cadena textual*. Las secuencias de escape sencillas (como `"\\"`, que es una barra diagonal inversa), las secuencias de escape hexadecimales (como `"\x0041"`, que es una A mayúscula) y las secuencias de escape Unicode (como `"\u0041"` que es una A mayúscula) se interpretan literalmente. Solo las secuencias de escape de comillas (`""`) no se interpretan literalmente, sino que generan comillas dobles. De igual modo, en el caso de una cadena interpolada literal, las secuencias de escape de llave (`{{` y `}}`) no se interpretan literalmente, sino que generan caracteres de llave simple. En el siguiente ejemplo se definen dos rutas de archivo idénticas, una mediante un literal de cadena normal y otra mediante el uso de un literal de cadena textual. Este es uno de los usos más comunes de los literales de cadena textual.

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//     c:\documents\files\u0066.txt
//     c:\documents\files\u0066.txt
```

En el ejemplo siguiente se muestra el efecto de definir un literal de cadena normal y un literal de cadena textual que contienen secuencias de caracteres idénticos.

```
string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//     He said, "This is the last chance!"
//     He said, "This is the last \u0063hance\x0021"
```

3. Para permitir que el compilador distinga entre los atributos en caso de conflicto de nomenclatura. Un atributo es una clase que deriva de Attribute. Normalmente, su nombre de tipo incluye el sufijo **Attribute**, aunque el compilador no exige el cumplimiento de esta convención. Es posible hacer referencia al atributo en el código mediante su nombre de tipo completo (por ejemplo, `[InfoAttribute]`) o mediante su nombre abreviado (por ejemplo, `[Info]`). Pero se produce un conflicto de nomenclatura si dos nombres abreviados de tipo de atributo son idénticos, y un nombre de tipo incluye el sufijo **Attribute** y el otro no. Por ejemplo, el código siguiente produce un error al compilarse porque el compilador no puede determinar si el atributo `Info` o `InfoAttribute` se aplica a la clase `Example`. Vea CS1614 para obtener más información.

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
   private string information;

   public Info(string info)
   {
      information = info;
   }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
   private string information;

   public InfoAttribute(string info)
   {
      information = info;
   }
}

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info'. Specify the full name 'InfoAttribute' to select it.
public class Example
{
   [InfoAttribute("The entry point.")]
   public static void Main()
   {
   }
}
```

Vea también

- Referencia de C#
- Guía de programación de C#
- Caracteres especiales de C#

Atributos reservados: atributos de nivel de ensamblado

22/04/2020 • 4 minutes to read • Edit Online

La mayoría de los atributos se aplican a elementos específicos del lenguaje, como las clases o los métodos, aunque algunos atributos son globales (se aplican a todo un ensamblado o módulo). Por ejemplo, el atributo AssemblyVersionAttribute se puede usar para insertar información de versión en un ensamblado, como en este ejemplo:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Los atributos globales aparecen en el código fuente después de cualquier directiva `using` de nivel superior y antes de cualquier declaración de tipo, módulo o espacio de nombres. Los atributos globales pueden aparecer en varios archivos de código fuente, pero estos archivos se deben compilar en un solo paso de compilación. Visual Studio agrega atributos globales al archivo AssemblyInfo.cs en proyectos de .NET Framework. Estos atributos no se agregan a los proyectos de .NET Core.

Los atributos de ensamblado son valores que proporcionan información sobre un ensamblado. Se dividen en las siguientes categorías:

- Atributos de identidad del ensamblado
- Atributos informativos
- Atributos de manifiesto del ensamblado

Atributos de identidad del ensamblado

Tres atributos, con un nombre seguro (si procede), determinan la identidad de un ensamblado: nombre, versión y referencia cultural. Estos atributos forman el nombre completo del ensamblado y son necesarios cuando se hace referencia a este en el código. Puede establecer la versión y la referencia cultural de un ensamblado mediante atributos, pero el valor de nombre lo establece el compilador, el IDE de Visual Studio en el cuadro de diálogo de información de ensamblado o la herramienta Assembly Linker (Al.exe) cuando se crea el ensamblado. El nombre del ensamblado se basa en el manifiesto del ensamblado. El atributo AssemblyFlagsAttribute especifica si pueden coexistir varias copias del ensamblado.

En la siguiente tabla se muestran los atributos de identidad.

ATRIBUTO	PROPÓSITO
AssemblyVersionAttribute	Especifica la versión de un ensamblado.
AssemblyCultureAttribute	Especifica la cultura que admite el ensamblado.
AssemblyFlagsAttribute	Especifica si un ensamblado admite la ejecución en paralelo en el mismo equipo, en el mismo proceso o en el mismo dominio de aplicación.

Atributos informativos

Puede usar atributos informativos para proporcionar información adicional de la empresa o el producto para un

ensamblado. En la tabla siguiente se muestran los atributos informativos definidos en el espacio de nombres System.Reflection.

ATRIBUTO	PROPÓSITO
AssemblyProductAttribute	Especifica un nombre de producto para un manifiesto del ensamblado.
AssemblyTrademarkAttribute	Especifica una marca comercial para un manifiesto del ensamblado.
AssemblyInformationalVersionAttribute	Especifica una versión informativa para un manifiesto del ensamblado.
AssemblyCompanyAttribute	Especifica un nombre de empresa para un manifiesto del ensamblado.
AssemblyCopyrightAttribute	Define un atributo personalizado que especifica un copyright para un manifiesto del ensamblado.
AssemblyFileVersionAttribute	Establece un número de versión específico para el recurso de versión de archivo Win32.
CLSCompliantAttribute	Indica si el ensamblado es compatible con Common Language Specification (CLS).

Atributos de manifiesto del ensamblado

Puede usar los atributos de manifiesto del ensamblado para proporcionar información en el manifiesto del ensamblado Los atributos incluyen el título, la descripción, el alias predeterminado y la configuración. En la tabla siguiente se muestran los atributos de manifiesto del ensamblado definidos en el espacio de nombres System.Reflection.

ATRIBUTO	PROPÓSITO
AssemblyTitleAttribute	Especifica un título de ensamblado para un manifiesto del ensamblado.
AssemblyDescriptionAttribute	Especifica una descripción de ensamblado para un manifiesto del ensamblado.
AssemblyConfigurationAttribute	Especifica una configuración de ensamblado (por ejemplo, comercial o depuración) para un manifiesto del ensamblado.
AssemblyDefaultAliasAttribute	Define un alias descriptivo predeterminado para un manifiesto del ensamblado.

Atributos reservados: ConditionalAttribute, ObsoleteAttribute, AttributeUsageAttribute

27/04/2020 • 9 minutes to read • Edit Online

Estos atributos se pueden aplicar a los elementos del código. Agregan significado semántico a esos elementos. El compilador usa esos significados semánticos para modificar su salida e informar de los posibles errores por parte de los desarrolladores que usan el código.

Atributo `Conditional`

El atributo `Conditional` hace que la ejecución de un método dependa de un identificador de preprocesamiento. El atributo `Conditional` es un alias de ConditionalAttribute y se puede aplicar a un método o a una clase de atributo.

En el ejemplo siguiente, `Conditional` se aplica a un método para habilitar o deshabilitar la representación de información de diagnóstico específica del programa:

```
#define TRACE_ON
using System;
using System.Diagnostics;

namespace AttributeExamples
{
    public class Trace
    {
        [Conditional("TRACE_ON")]
        public static void Msg(string msg)
        {
            Console.WriteLine(msg);
        }
    }

    public class TraceExample
    {
        public static void Main()
        {
            Trace.Msg("Now in Main...");
            Console.WriteLine("Done.");
        }
    }
}
```

Si no se define el identificador `TRACE_ON`, no se muestra el resultado del seguimiento. Explore por sí mismo en la ventana interactiva.

El atributo `Conditional` se suele usar con el identificador `DEBUG` para habilitar las funciones de seguimiento y de registro para las compilaciones de depuración, pero no en las compilaciones de versión, como se muestra en el ejemplo siguiente:

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

Al llamar a un método marcado como condicional, la presencia o ausencia del símbolo de preprocesamiento

especificado determina si se incluye o se omite la llamada. Si el símbolo está definido, se incluye la llamada; de lo contrario, se omite la llamada. Un método condicional debe ser un método de una declaración de clase o estructura, y no debe tener un tipo de valor devuelto `void`. El uso de `Conditional` resulta más limpio y elegante, y menos propenso a generar errores que incluir los métodos dentro de bloques `#if…#endif`.

Si un método tiene varios atributos `Conditional`, se incluye una llamada al método si se define uno o más símbolos condicionales (los símbolos se vinculan de manera lógica entre sí mediante el operador OR). En el ejemplo siguiente, la presencia de `A` o `B` da como resultado una llamada al método:

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Uso de `Conditional` con clases de atributos

El atributo `Conditional` también se puede aplicar a una definición de clase de atributo. En el ejemplo siguiente, el atributo personalizado `Documentation` solo agregará información a los metadatos si se define `DEBUG`.

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

Atributo `Obsolete`

El atributo `Obsolete` marca un elemento de código como ya no recomendado para su uso. El uso de una entidad marcada como obsoleta genera una advertencia o un error. El atributo `Obsolete` es un atributo de uso único y se puede aplicar a cualquier entidad que admita atributos. `Obsolete` es un alias de ObsoleteAttribute.

En el ejemplo siguiente, el atributo `Obsolete` se aplica a la clase `A` y al método `B.OldMethod`. Dado que el segundo argumento del constructor de atributos aplicado a `B.OldMethod` está establecido en `true`, este método producirá un error del compilador, mientras que, si se usa la clase `A`, solo se generará una advertencia. En cambio, si se llama a `B.NewMethod`, no se generará ninguna advertencia o error. Por ejemplo, al usarla con las definiciones anteriores, el código siguiente genera dos advertencias y un error:

```
using System;

namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }
    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}
```

La cadena proporcionada como primer argumento al constructor del atributo se mostrará como parte de la advertencia o el error. Se generan dos advertencias para la clase `A`: una para la declaración de la referencia de clase y otra para el constructor de clases. El atributo `Obsolete` se puede usar sin argumentos, pero se recomienda incluir una explicación de qué se debe usar en su lugar.

Atributo `AttributeUsage`

El atributo `AttributeUsage` determina cómo se puede usar una clase de atributo personalizado. AttributeUsageAttribute es un atributo que se aplica a las definiciones de atributo personalizado. El atributo `AttributeUsage` permite controlar lo siguiente:

- Los atributos de elementos de programa que se pueden aplicar. A menos que el uso esté restringido, un atributo se puede aplicar a cualquiera de los siguientes elementos de programa:
  - ensamblado
  - module
  - campo
  - event
  - método
  - param
  - propiedad
  - return
  - type
- Si un atributo se puede aplicar a un mismo elemento de programa varias veces.

- Si las clases derivadas heredan atributos.

La configuración predeterminada se parece al siguiente ejemplo cuando se aplica explícitamente:

```
[AttributeUsage(AttributeTargets.All,
                   AllowMultiple = false,
                   Inherited = true)]
class NewAttribute : Attribute { }
```

En este ejemplo, la clase `NewAttribute` se puede aplicar a cualquier elemento de programación compatible, pero solamente se puede aplicar una vez a cada entidad. Las clases derivadas heredan el atributo cuando se aplica a una clase base.

Los argumentos AllowMultiple y Inherited son opcionales, por lo que el siguiente código tiene el mismo efecto:

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

El primer argumento AttributeUsageAttribute debe ser uno o varios elementos de la enumeración AttributeTargets. Se pueden vincular diversos tipos de destino con el operador OR, como se refleja en el siguiente ejemplo:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

A partir de C# 7.3, los atributos se pueden aplicar a la propiedad o el campo de respaldo de una propiedad implementada automáticamente. El atributo se aplica a la propiedad, a menos que se indique el especificador `field` en el atributo. Ambos se muestran en el siguiente ejemplo:

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; }

    // Attribute attached to backing field:
    [field:NewPropertyOrField]
    public string Description { get; set; }
}
```

Si el argumento AllowMultiple es `true`, el atributo resultante se puede aplicar más de una vez a cada una de las entidades, como se muestra en el siguiente ejemplo:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

En este caso, `MultiUseAttribute` se puede aplicar varias veces porque `AllowMultiple` está establecido en `true`. Los dos formatos mostrados para aplicar varios atributos son válidos.

Si Inherited se establece en `false`, las clases que se derivan de una clase con atributos no heredan el atributo. Por

ejemplo:

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

En este caso, `NonInheritedAttribute` no se aplica a `DClass` a través de la herencia.

Vea también

- Attribute
- System.Reflection
- Atributos
- Reflexión

Atributos reservados: determinación de la información del autor de la llamada

22/04/2020 • 5 minutes to read • Edit Online

Mediante los atributos de información, se obtiene información sobre el autor de la llamada a un método. Se obtiene la ruta de acceso al código fuente, el número de línea del código fuente y el nombre de miembro del autor de la llamada. Para obtener la información del llamador del miembro, use los atributos que se aplican a los parámetros opcionales. Cada parámetro opcional especifica un valor predeterminado. En la tabla siguiente se enumeran los atributos de información del llamador que se definen en el espacio de nombres System.Runtime.CompilerServices:

ATRIBUTO	DESCRIPCIÓN	TIPO
CallerFilePathAttribute	Ruta de acceso completa del archivo de código fuente que contiene el llamador. La ruta de acceso completa es la ruta de acceso en tiempo de compilación.	`String`
CallerLineNumberAttribute	Número de línea del archivo de código fuente desde el que se llama al método.	`Integer`
CallerMemberNameAttribute	Nombre de método o de propiedad del llamador.	`String`

Esta información facilita la creación de herramientas de seguimiento, depuración y diagnóstico. En el ejemplo siguiente se muestra cómo usar atributos de información del autor de la llamada. En cada llamada al método `TraceMessage`, la información del llamador se sustituye como argumentos para los parámetros opcionales.

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
//  source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

Debe especificar un valor predeterminado explícito para cada parámetro opcional. No puede aplicar atributos de información del autor de la llamada a los parámetros que no se especifican como opcionales. Los atributos de información del autor de la llamada no crean un parámetro opcional, sino que influyen en el valor predeterminado

que se pasa cuando se omite el argumento. Los valores de información del autor de la llamada se emiten como literales en el lenguaje intermedio (IL) en tiempo de compilación. A diferencia de los resultados de la propiedad StackTrace para las excepciones, los resultados no se ven afectados por confusión. Puede proporcionar explícitamente los argumentos opcionales para controlar la información del llamador u ocultarla.

Nombres de miembro

Se puede utilizar el atributo `CallerMemberName` para evitar especificar el nombre de miembro como un argumento `String` para el método llamado. Mediante esta técnica, se evita el problema de que la **refactorización de cambio de nombre** no cambie los valores `String`. Esta ventaja es especialmente útil para las siguientes tareas:

- Usar el seguimiento y las rutinas de diagnóstico.
- Implementar la interfaz INotifyPropertyChanged al enlazar datos. Esta interfaz permite que la propiedad de un objeto notifique a un control enlazado que la propiedad ha cambiado, de forma que el control pueda mostrar información actualizada. Sin el atributo `CallerMemberName`, se debe especificar el nombre de propiedad como un literal.

En el gráfico siguiente se muestran los nombres de miembro que se devuelven cuando se utiliza el atributo `CallerMemberName`.

LAS LLAMADAS SE PRODUCEN EN	RESULTADO DEL NOMBRE DE MIEMBRO
Método, propiedad o evento	Nombre del método, propiedad o evento en el que se originó la llamada.
Constructor	Cadena ".ctor"
Constructor estático	Cadena ".cctor"
Destructor	Cadena "Finalize"
Conversiones u operadores definidos por el usuario	Nombre generado para el miembro, por ejemplo, "op_Addition".
Constructor de atributo	Nombre del método o la propiedad a la que se aplica el atributo. Si el atributo es cualquier elemento dentro de un miembro (como un parámetro, un valor devuelto o un parámetro de tipo genérico), el resultado es el nombre del miembro asociado a este elemento.
Ningún miembro contenedor (por ejemplo, nivel de ensamblado o atributos que se aplican a tipos)	El valor predeterminado del parámetro opcional.

Vea también

- Argumentos opcionales y con nombre
- System.Reflection
- Attribute
- Atributos

Los atributos reservados contribuyen al análisis estático del estado NULL del compilador.

27/04/2020 • 28 minutes to read • Edit Online

En un contexto que admite un valor NULL, el compilador realiza un análisis estático del código para determinar el estado NULL de todas las variables de tipo de referencia:

- *not null*: el análisis estático ha determinado que a la variable se le asigna un valor distinto de NULL.
- *maybe null*: el análisis estático no puede determinar que a la variable se le asigna un valor distinto de NULL.

Puede aplicar una serie de atributos que proporcionan información al compilador sobre la semántica de las API. Esa información ayuda al compilador a realizar análisis estáticos y a determinar si una variable no es NULL. En este artículo se proporciona una breve descripción de cada uno de esos atributos y cómo usarlos. En todos los ejemplos se asume el uso de C# 8.0 o una versión más reciente, y que el código se encuentra en un contexto que admite un valor NULL.

Para empezar, se usará un ejemplo conocido. Imagine que la biblioteca tiene la siguiente API para recuperar una cadena de recursos:

```
bool TryGetMessage(string key, out string message)
```

En el ejemplo anterior se sigue el conocido patrón de `Try*` en .NET. Hay dos argumentos de referencia para esta API: los parámetros `key` y `message`. Esta API tiene las siguientes reglas relacionadas con la obtención del valor NULL de estos argumentos:

- Los autores de la llamada no deben pasar `null` como argumento para `key`.
- Los autores de la llamada pueden pasar una variable cuyo valor sea `null` como argumento de `message`.
- Si el método `TryGetMessage` devuelve `true`, el valor de `message` no es NULL. Si el valor devuelto es `false,`, el valor de `message` (y su estado NULL) es NULL.

La regla para `key` se puede expresar mediante el tipo de variable: `key` debe ser un tipo de referencia que no acepte valores NULL. El parámetro `message` es más complejo. Permite `null` como argumento, pero garantiza que, si se ejecuta correctamente, el argumento `out` no sea NULL. En estos escenarios, necesita un vocabulario más completo para describir las expectativas.

Se han agregado varios atributos para expresar información adicional sobre el estado NULL de las variables. Todo el código escrito antes de que C# 8 introdujera los tipos de referencia que aceptan valores NULL *desconocía los valores NULL*. Esto significa que es posible que cualquier variable de tipo de referencia sea NULL, pero no se necesitan comprobaciones de valores NULL. Una vez que el código *admite valores NULL*, esas reglas cambian. Los tipos de referencia nunca deben ser del valor `null`, y los que admitan valores NULL se deben comprobar con `null` antes de desreferenciarlos.

Es probable que las reglas de las API sean más complicadas, como se ha visto en el escenario de la API `TryGetValue`. Muchas de las API tienen reglas más complejas para cuando las variables pueden ser `null` o no. En estos casos, usará uno de los atributos siguientes para expresar estas reglas:

- AllowNull: un argumento de entrada que no acepta valores NULL puede ser NULL.
- DisallowNull: un argumento de entrada que admite un valor NULL nunca debe ser NULL.
- MaybeNull: un valor devuelto que no acepta valores NULL puede ser NULL.

- NotNull: un valor devuelto que admite un valor NULL nunca será NULL.
- MaybeNullWhen: un argumento de entrada que no acepta valores NULL puede ser NULL cuando el método devuelve el valor `bool` especificado.
- NotNullWhen: un argumento de entrada que admite un valor NULL nunca será NULL cuando el método devuelve el valor `bool` especificado.
- NotNullIfNotNull: un valor devuelto no es NULL si el argumento del parámetro especificado no es NULL.
- DoesNotReturn: un método nunca devuelve un valor. Es decir, siempre inicia una excepción.
- DoesNotReturnIf: este método nunca devuelve un valor si el parámetro `bool` asociado tiene el valor especificado.

Las descripciones anteriores son una referencia rápida a lo que hace cada atributo. En cada una de las secciones siguientes se describen el comportamiento y el significado con más detalle.

Al agregar estos atributos, se proporciona más información al compilador sobre las reglas de la API. Cuando el código que realiza la llamada se compila en un contexto habilitado para aceptar valores NULL, el compilador advertirá a los autores de la llamada cuando infrinjan esas reglas. Estos atributos no habilitan comprobaciones adicionales en la implementación.

Especificación de condiciones previas: `AllowNull` y `DisallowNull`

Considere una propiedad de lectura y escritura que nunca devuelve `null` porque tiene un valor predeterminado razonable. Los autores de la llamada pasan `null` al descriptor de acceso set cuando lo establecen en el valor predeterminado. Por ejemplo, imagine un sistema de mensajería que solicita un nombre de pantalla en un salón de chat. Si no se proporciona ninguno, el sistema genera uno aleatorio:

```
public string ScreenName
{
   get => screenName;
   set => screenName = value ?? GenerateRandomScreenName();
}
private string screenName;
```

Al compilar el código anterior en un contexto en el que se desconocen los valores NULL, todo es correcto. Una vez que se habilitan los tipos de referencia que admiten un valor NULL, la propiedad `ScreenName` se convierte en una referencia que no acepta valores NULL. Eso es correcto para el descriptor de acceso `get`: nunca devuelve `null`. No es necesario que los autores de la llamada comprueben `null` en la propiedad devuelta. Pero ahora, al establecer la propiedad en `null`, se genera una advertencia. Para continuar admitiendo este tipo de código, agregue el atributo System.Diagnostics.CodeAnalysis.AllowNullAttribute a la propiedad, como se muestra en el código siguiente:

```
[AllowNull]
public string ScreenName
{
   get => screenName;
   set => screenName = value ?? GenerateRandomScreenName();
}
private string screenName = GenerateRandomScreenName();
```

Es posible que tenga que agregar una directiva `using` para System.Diagnostics.CodeAnalysis a fin de usar este y otros atributos descritos en este artículo. El atributo se aplica a la propiedad, no al descriptor de acceso `set`. El atributo `AllowNull` especifica *condiciones previas* y solo se aplica a las entradas. El descriptor de acceso `get` tiene un valor devuelto, pero no tiene argumentos de entrada. Por tanto, el atributo `AllowNull` solo se aplica al descriptor de acceso `set`.

En el ejemplo anterior se muestra qué se debe buscar al agregar el atributo `AllowNull` en un argumento:

1. El contrato general para esa variable es que no debe ser `null`, por lo que quiere un tipo de referencia que no acepte valores NULL.
2. Hay escenarios para que la variable de entrada sea `null`, aunque no sean el uso más común.

En la mayoría de los casos necesitará este atributo para las propiedades, o bien para los argumentos `in`, `out` y `ref`. El atributo `AllowNull` es la mejor opción cuando una variable normalmente no es NULL, pero debe permitir `null` como condición previa.

Compare esto con los escenarios para usar `DisallowNull`: este atributo se usa para especificar que una variable de entrada de un tipo de referencia que admite un valor NULL no debe ser `null`. Considere una propiedad donde `null` es el valor predeterminado, pero los clientes solo pueden establecerla en un valor que no sea NULL. Observe el código siguiente:

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

El código anterior es la mejor manera de expresar el diseño de que `ReviewComment` pueda ser `null`, pero no se puede establecer en `null`. Una vez que este código admita valores NULL, puede expresar este concepto con más claridad para los autores de la llamada mediante System.Diagnostics.CodeAnalysis.DisallowNullAttribute:

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

En un contexto que admite un valor NULL, el descriptor de acceso `get` de `ReviewComment` podría devolver el valor predeterminado de `null`. El compilador advierte que se debe comprobar antes del acceso. Además, advierte a los autores de la llamada que, aunque podría ser `null`, no deberían establecerlo de forma explícita en `null`. El atributo `DisallowNull` también especifica una *condición previa*, no afecta al descriptor de acceso `get`. Use el atributo `DisallowNull` cuando observe estas características:

1. La variable podría ser `null` en escenarios principales, a menudo cuando se crea su primera instancia.
2. La variable no se debe establecer de forma explícita en `null`.

Estas situaciones son comunes en el código en el que originalmente *se desconocían los valores NULL*. Es posible que las propiedades de objeto se establezcan en dos operaciones de inicialización distintas. Es posible que algunas propiedades se establezcan solo después de que se haya completado algún trabajo asincrónico.

Los atributos `AllowNull` y `DisallowNull` permiten especificar que las condiciones previas de las variables puedan no coincidir con las anotaciones que admiten un valor NULL en esas variables. Proporcionan más detalles sobre las características de la API. Esta información adicional ayuda a los autores de la llamada a usar la API de manera correcta. Recuerde que debe especificar las condiciones previas mediante los atributos siguientes:

- AllowNull: un argumento de entrada que no acepta valores NULL puede ser NULL.
- DisallowNull: un argumento de entrada que admite un valor NULL nunca debe ser NULL.

Especificación de condiciones posteriores: `MaybeNull` y `NotNull`

Imagine que tiene un método con la siguiente firma:

```
public Customer FindCustomer(string lastName, string firstName)
```

Probablemente haya escrito un método como este para devolver `null` cuando no se encuentra el nombre que se busca. `null` indica claramente que no se ha encontrado el registro. En este ejemplo, es probable que cambie el tipo de valor devuelto de `Customer` a `Customer?`. Al declarar el valor devuelto como un tipo de referencia que admite un valor NULL, se especifica claramente la intención de esta API.

Por los motivos descritos en Definiciones genéricas y nulabilidad, esa técnica no funciona con los métodos genéricos. Puede tener un método genérico que siga un patrón similar:

```
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> match)
```

No puede especificar que el valor devuelto sea `T?`. El método devuelve `null` cuando no se encuentra el elemento buscado. Como no puede declarar un tipo de valor devuelto `T?`, agregue la anotación `MaybeNull` al valor devuelto del método:

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> match)
```

El código anterior informa a los autores de la llamada de que el contrato implica un tipo que no acepta valores NULL, pero el valor devuelto *puede* realmente ser NULL. Use el atributo `MaybeNull` cuando la API deba ser un tipo que no acepta valores NULL, normalmente un parámetro de tipo genérico, pero puede haber casos en los que se devuelva `null`.

También puede especificar que un valor devuelto o un argumento `out` o `ref` no sea NULL aunque el tipo sea un tipo de referencia que admite un valor NULL. Considere un método que garantiza que una matriz es lo suficientemente grande como para contener un número de elementos. Si el argumento de entrada no tiene capacidad, la rutina asignará una nueva matriz y copiará todos los elementos existentes en ella. Si el argumento de entrada es `null`, la rutina asignará un almacenamiento nuevo. Si hay capacidad suficiente, la rutina no hace nada:

```
public void EnsureCapacity<T>(ref T[] storage, int size)
```

Puede llamar a esta rutina de esta forma:

```
// messages has the default value (null) when EnsureCapacity is called:
EnsureCapacity<string>(ref messages, 10);
// messages is not null.
EnsureCapacity<string>(messages, 50);
```

Después de habilitar los tipos de referencia NULL, querrá asegurarse de que el código anterior se compila sin advertencias. Cuando el método devuelve un valor, se garantiza que el argumento `storage` no es NULL. Pero es aceptable llamar a `EnsureCapacity` con una referencia nula. Puede convertir a `storage` en un tipo de referencia que acepte valores NULL y agregar la condición posterior `NotNull` a la declaración del parámetro:

```
public void EnsureCapacity<T>([NotNull]ref T[]? storage, int size)
```

En el código anterior se expresa con claridad el contrato existente: Los autores de la llamada pueden pasar una variable con el valor `null`, pero se garantiza que el valor devuelto nunca será NULL. El atributo `NotNull` es muy útil para los argumentos `ref` y `out`, donde `null` se puede pasar como argumento, pero se garantiza que ese argumento no será NULL cuando el método devuelva un valor.

Las condiciones posteriores condicionales se especifican mediante los atributos siguientes:

- MaybeNull: un valor devuelto que no acepta valores NULL puede ser NULL.
- NotNull: un valor devuelto que admite un valor NULL nunca será NULL.

Especificación de condiciones posteriores condicionales: `NotNullWhen`, `MaybeNullWhen` y `NotNullIfNotNull`

Es probable que esté familiarizado con el método `string` de String.IsNullOrEmpty(String). Este método devuelve `true` cuando el argumento es NULL o una cadena vacía. Es una forma de comprobación de valores NULL: No es necesario que los autores de la llamada comprueben los valores NULL del argumento si el método devuelve `false`. Para hacer que un método como este admita valores NULL, tendría que establecer el argumento en un tipo de referencia que admite un valor NULL y agregar el atributo `NotNullWhen`:

```
bool IsNullOrEmpty([NotNullWhen(false)]string? value);
```

Esto informa al compilador de que no es necesario comprobar los valores NULL en el código cuyo valor devuelto sea `false`. La adición del atributo informa al análisis estático del compilador que `IsNullOrEmpty` realiza la comprobación de valores NULL necesaria: cuando devuelve `false`, el argumento de entrada no es `null`.

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
   int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

El método String.IsNullOrEmpty(String) se anotará como se ha mostrado antes para .NET Core 3.0. Es posible que tenga métodos similares en el código base que comprueben valores NULL en el estado de los objetos. El compilador no reconocerá los métodos de comprobación de valores NULL personalizados y tendrá que agregar personalmente las anotaciones. Al agregar el atributo, el análisis estático del compilador sabe cuándo se han comprobado los valores NULL en la variable.

Otro uso de estos atributos es el patrón `Try*`. Las condiciones posteriores para las variables `ref` y `out` se comunican a través del valor devuelto. Observe este método mostrado antes:

```
bool TryGetMessage(string key, out string message)
```

El método anterior sigue una expresión de .NET típica: el valor devuelto indica si `message` se ha establecido en el valor encontrado o, si no se ha encontrado ningún mensaje, en el valor predeterminado. Si el método devuelve `true`, el valor de `message` no es NULL; de lo contrario, el método establece `message` en NULL.

Puede comunicar esa expresión mediante el atributo `NotNullWhen`. Al actualizar la firma para los tipos de referencia que admiten un valor NULL, convierta `message` en `string?` y agregue un atributo:

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
```

En el ejemplo anterior, se sabe que el valor de `message` no es NULL cuando `TryGetMessage` devuelve `true`. Debe anotar de la misma forma los métodos similares en el código base: los argumentos pueden ser `null`, y se sabe que no son NULL cuando el método devuelve `true`.

Hay un atributo final que también puede necesitar. En ocasiones, el estado NULL de un valor devuelto depende del estado NULL de uno o más argumentos de entrada. Estos métodos devolverán un valor no NULL siempre que determinados argumentos de entrada no sean `null`. Para anotar correctamente estos métodos, use el atributo `NotNullIfNotNull`. Observe el método siguiente:

```
string GetTopLevelDomainFromFullUrl(string url);
```

Si el argumento `url` no es NULL, el resultado no es `null`. Una vez que se hayan habilitado las referencias nulas, esa firma funcionará correctamente, siempre que la API no acepte nunca una entrada NULL. Pero si la entrada puede ser NULL, el valor devuelto también podría serlo. Por tanto, podría cambiar la firma por el código siguiente:

```
string? GetTopLevelDomainFromFullUrl(string? url);
```

Esto también funciona, pero a menudo obliga a los autores de la llamada a implementar comprobaciones de `null` adicionales. El contrato es que el valor devuelto solo será `null` cuando el argumento de entrada `url` sea `null`. Para expresar ese contrato, tendría que anotar este método como se muestra en el código siguiente:

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url);
```

El valor devuelto y el argumento se han anotado con `?`, lo que indica que cualquiera podría ser `null`. El atributo aclara todavía más que el valor devuelto no será NULL cuando el argumento `url` no sea `null`.

Las condiciones posteriores condicionales se especifican mediante estos atributos:

- MaybeNullWhen: un argumento de entrada que no acepta valores NULL puede ser NULL cuando el método devuelve el valor `bool` especificado.
- NotNullWhen: un argumento de entrada que admite un valor NULL nunca será NULL cuando el método devuelva el valor `bool` especificado.
- NotNullIfNotNull: un valor devuelto no es NULL si el argumento de entrada del parámetro especificado no es NULL.

Comprobación de código inaccesible

Algunos métodos, normalmente los asistentes de excepciones u otros métodos de utilidad, siempre salen mediante el inicio de una excepción. O bien, un asistente puede iniciar una excepción basada en el valor de un argumento booleano.

En el primer caso, puede agregar el atributo `DoesNotReturn` a la declaración del método. El compilador le ayuda de tres maneras. En primer lugar, emite una advertencia si hay una ruta de acceso por la que el método puede salir sin iniciar una excepción. En segundo lugar, el compilador marca cualquier código después de una llamada a ese método como *inaccesible*, hasta que se encuentre una cláusula `catch` adecuada. Por último, el código inaccesible no afectará a ningún estado NULL. Considere este método:

```
[DoesNotReturn]
private void FailFast()
{
   throw new InvalidOperationException();
}

public void SetState(object containedField)
{
   if (!isInitialized)
      FailFast();

   // unreachable code:
   this.field = containedField;
}
```

En el segundo caso, se agrega el atributo `DoesNotReturnIf` a un parámetro booleano del método. Puede modificar el ejemplo anterior de esta manera:

```
private void FailFast([DoesNotReturnIf(false)]bool isValid)
{
   if (!isValid)
       throw new InvalidOperationException();
}

public void SetState(object containedField)
{
   FailFast(isInitialized);

   // unreachable code when "isInitialized" is false:
   this.field = containedField;
}
```

Resumen

Agregar tipos de referencia que aceptan valores NULL proporciona un vocabulario inicial para describir las expectativas de las API para las variables que podrían ser `null`. Los atributos adicionales proporcionan un vocabulario más completo para describir el estado NULL de las variables como condiciones previas y posteriores. Estos atributos describen con más claridad las expectativas y proporcionan una mejor experiencia para los desarrolladores que usan las API.

A medida que actualice las bibliotecas para un contexto que admite un valor NULL, agregue estos atributos para guiar a los usuarios de las API al uso correcto. Estos atributos ayudan a describir de forma completa el estado NULL de los argumentos de entrada y los valores devueltos:

- AllowNull: un argumento de entrada que no acepta valores NULL puede ser NULL.
- DisallowNull: un argumento de entrada que admite un valor NULL nunca debe ser NULL.
- MaybeNull: un valor devuelto que no acepta valores NULL puede ser NULL.
- NotNull: un valor devuelto que admite un valor NULL nunca será NULL.
- MaybeNullWhen: un argumento de entrada que no acepta valores NULL puede ser NULL cuando el método devuelve el valor `bool` especificado.
- NotNullWhen: un argumento de entrada que admite un valor NULL nunca será NULL cuando el método devuelva el valor `bool` especificado.
- NotNullIfNotNull: un valor devuelto no es NULL si el argumento de entrada del parámetro especificado no es NULL.
- DoesNotReturn: un método nunca devuelve un valor. Es decir, siempre inicia una excepción.
- DoesNotReturnIf: este método nunca devuelve un valor si el parámetro `bool` asociado tiene el valor

especificado.

Directivas de preprocesador de C#

18/03/2020 • 2 minutes to read • Edit Online

Esta sección contiene información estas directivas de preprocesador de C#:

- #if
- #else
- #elif
- #endif
- #define
- #undef
- #warning
- #error
- #line
- #region
- #endregion
- #pragma
- #pragma warning
- #pragma checksum

Vea cada uno de los temas para obtener más información y ejemplos.

Aunque el compilador no tiene un preprocesador independiente, las directivas descritas en esta sección se procesan como si hubiera uno. Se usan para facilitar la compilación condicional. A diferencia de las directivas de C y C++, no se pueden usar estas directivas para crear macros.

Una directiva de preprocesador debe ser la única instrucción en una línea.

Vea también

- Referencia de C#
- Guía de programación de C#

#if (referencia de C#)

18/03/2020 • 5 minutes to read • Edit Online

Cuando el compilador de C# encuentra una directiva `#if`, seguida finalmente por una directiva #endif, compila el código entre las directivas solo si se ha definido el símbolo especificado. A diferencia de C y C++, no se puede asignar un valor numérico a un símbolo. La instrucción `#if` en C# es booleana y solo comprueba si el símbolo se ha definido o no. Por ejemplo:

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

Puede usar los operadores == (igualdad) y != (desigualdad) solo para comprobar los valores booleanos `true` o `false`. `true` significa que el símbolo está definido. La instrucción `#if DEBUG` tiene el mismo significado que `#if (DEBUG == true)`. Puede usar los operadores && (AND), || (OR) y ! (NOT) para evaluar si se han definido varios símbolos. Es posible agrupar símbolos y operadores mediante paréntesis.

Comentarios

`#if`, junto con las directivas #else, #elif, #endif, #define y #undef, permite incluir o excluir código basado en la existencia de uno o varios símbolos. Esto puede resultar útil al compilar código para una compilación de depuración o al compilar para una configuración específica.

Una directiva condicional que empieza con una directiva `#if` debe terminar de forma explícita con una directiva `#endif`.

`#define` permite definir un símbolo. Si luego se usa el símbolo como la expresión pasada a la directiva `#if`, la expresión se evalúa como `true`.

También se puede definir un símbolo con la opción del compilador -define. La definición de un símbolo se puede anular mediante #undef.

Un símbolo definido con `-define` o `#define` no debe entrar en conflicto con una variable del mismo nombre. Es decir, un nombre de variable no debe pasarse a una directiva de preprocesador y un símbolo solo puede ser evaluado por una directiva de preprocesador.

El ámbito de un símbolo creado con `#define` es el archivo en que se ha definido.

El sistema de compilación también tiene en cuenta los símbolos de preprocesador predefinidos que representan distintos marcos de destino en proyectos de estilo SDK. Resultan útiles al crear aplicaciones que pueden tener como destino más de una versión o implementación de .NET.

VERSIONES DE .NET FRAMEWORK DE DESTINO	SÍMBOLOS
.NET Framework	`NETFRAMEWORK`, `NET20`, `NET35`, `NET40`, `NET45`, `NET451`, `NET452`, `NET46`, `NET461`, `NET462`, `NET47`, `NET471`, `NET472`, `NET48`

VERSIONES DE .NET FRAMEWORK DE DESTINO	SÍMBOLOS
.NET Standard	`NETSTANDARD`, `NETSTANDARD1_0`, `NETSTANDARD1_1`, `NETSTANDARD1_2`, `NETSTANDARD1_3`, `NETSTANDARD1_4`, `NETSTANDARD1_5`, `NETSTANDARD1_6`, `NETSTANDARD2_0`, `NETSTANDARD2_1`
.NET Core	`NETCOREAPP`, `NETCOREAPP1_0`, `NETCOREAPP1_1`, `NETCOREAPP2_0`, `NETCOREAPP2_1`, `NETCOREAPP2_2`, `NETCOREAPP3_0`, `NETCOREAPP3_1`

NOTE

En el caso de los proyectos de .NET Framework tradicionales, tendrá que configurar manualmente los símbolos de compilación condicional para las diferentes plataformas de destino en Visual Studio a través de las páginas de propiedades del proyecto.

Otros símbolos predefinidos incluyen las constantes DEBUG y TRACE. Puede invalidar los valores establecidos para el proyecto con `#define`. Por ejemplo, el símbolo DEBUG se establece automáticamente según las propiedades de configuración de compilación (modo de "depuración" o de "versión").

Ejemplos

En el ejemplo siguiente se muestra cómo definir un símbolo MYTEST en un archivo y luego probar los valores de los símbolos MYTEST y DEBUG. El resultado de este ejemplo depende de si ha compilado el proyecto en modo de configuración de depuración o de versión.

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

En el ejemplo siguiente se muestra cómo probar distintos marcos de destino para que se puedan usar las API más recientes cuando sea posible:

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#
- Compilación condicional con Trace y Debug

#else (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#else` permite crear una directiva condicional compuesta, de modo que, si ninguna de las expresiones de las directivas #if o #elif (opcional) anteriores se evalúan como `true`, el compilador evaluará todo el código entre `#else` y la directiva `#endif` siguiente.

Comentarios

#endif debe ser la siguiente directiva de preprocesador después de `#else`. Vea #if para obtener un ejemplo de cómo usar `#else`.

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#elif (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#elif` permite crear una directiva condicional compuesta. La expresión `#elif` se evaluará si ninguna de las expresiones de las directivas #if o `#elif` (opcional) precedentes se evalúan como `true`. Si una expresión `#elif` se evalúa como `true`, el compilador incluye en la compilación todo el código comprendido entre `#elif` y la siguiente directiva condicional. Por ejemplo:

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

Se pueden usar los operadores `==` (igualdad), `!=` (desigualdad), `&&` (y), así como `||` (o), para evaluar varios símbolos. Es posible agrupar símbolos y operadores mediante paréntesis.

Comentarios

`#elif` equivale a usar:

```
#else
#if
```

El uso de `#elif` es más simple ya que cada directiva `#if` requiere una directiva #endif, mientras que una directiva `#elif` se puede usar sin la directiva `#endif` correspondiente.

Vea #if para obtener un ejemplo de cómo usar `#elif`.

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#endif (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#endif` especifica el final de una directiva condicional, que comienza con la directiva #if. Por ejemplo,

```
#define DEBUG
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

Comentarios

Una directiva condicional que empieza con una directiva `#if` debe terminar de forma explícita con una directiva `#endif`. Vea #if para obtener un ejemplo de cómo usar `#endif`.

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#define (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

Usa `#define` para definir un símbolo. Si usa el símbolo como expresión que se pasa a la directiva #if, la expresión se evaluará como `true`, como se muestra en el siguiente ejemplo:

```
#define DEBUG
```

Comentarios

> **NOTE**
> La directiva `#define` no puede usarse para declarar valores constantes como suele hacerse en C y C++. En C#, las constantes se definen mejor como miembros estáticos de una clase o struct. Si tiene varias constantes de este tipo, puede considerar la posibilidad de crear una clase "Constants" independiente donde incluirlas.

Los símbolos se pueden usar para especificar condiciones de compilación. Puede comprobar el símbolo tanto con #if como con #elif. También se puede usar ConditionalAttribute para realizar una compilación condicional.

Es posible definir un símbolo, pero no asignar un valor a un símbolo. La directiva `#define` debe aparecer en el archivo antes de que use cualquier instrucción que tampoco sea una directiva del preprocesador.

También se puede definir un símbolo con la opción del compilador -define. La definición de un símbolo se puede anular mediante #undef.

Un símbolo definido mediante `-define` o `#define` no debe entrar en conflicto con una variable del mismo nombre. Es decir, no se debería pasar un nombre de variable a una directiva de preprocesador ni evaluar solo un símbolo mediante una directiva de preprocesador.

El ámbito de un símbolo que se ha creado mediante `#define` corresponde al archivo en el que se ha definido.

Como se muestra en el siguiente ejemplo, debe colocar directivas `#define` en la parte superior del archivo.

```
#define DEBUG
//#define TRACE
#undef TRACE

using System;

public class TestDefine
{
    static void Main()
    {
#if (DEBUG)
        Console.WriteLine("Debugging is enabled.");
#endif

#if (TRACE)
     Console.WriteLine("Tracing is enabled.");
#endif
    }
}
// Output:
// Debugging is enabled.
```

Para obtener un ejemplo de cómo anular la definición de un símbolo, vea #undef.

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#
- const
- Compilación condicional con Trace y Debug
- #undef
- #if

#undef (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#undef` le permite anular la definición de un símbolo, de tal forma que, si se usa como la expresión en una directiva #if, la expresión se evaluará como `false`.

Un símbolo se puede definir con la directiva #define o la opción del compilador -define. La directiva `#undef` debe aparecer en el archivo antes de que use cualquier instrucción que tampoco sea una directiva.

Ejemplo

```
// preprocessor_undef.cs
// compile with: /d:DEBUG
#undef DEBUG
using System;
class MyClass
{
    static void Main()
    {
#if DEBUG
        Console.WriteLine("DEBUG is defined");
#else
        Console.WriteLine("DEBUG is not defined");
#endif
    }
}
```

DEBUG está sin definirse

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#warning (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#warning` permite generar una advertencia del compilador CS1030 de nivel uno desde una ubicación específica en el código. Por ejemplo:

```
#warning Deprecated code in this method.
```

Comentarios

Un uso común de `#warning` es en una directiva condicional. También es posible generar un error definido por el usuario con #error.

Ejemplo

```
// preprocessor_warning.cs
// CS1030 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#warning DEBUG is defined
#endif
    }
}
```

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#error (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#error` permite generar un error CS1029 definido por el usuario desde una ubicación específica en el código. Por ejemplo:

```
#error Deprecated code in this method.
```

Comentarios

Un uso común de `#error` es en una directiva condicional.

También es posible generar una advertencia definida por el usuario con #warning.

Ejemplo

```
// preprocessor_error.cs
// CS1029 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#error DEBUG is defined
#endif
    }
}
```

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#line (Referencia de C#)

18/03/2020 • 4 minutes to read • Edit Online

`#line` le permite modificar el número de línea del compilador y (opcionalmente) la salida del nombre de archivo de errores y advertencias.

En el siguiente ejemplo, se muestra cómo notificar dos advertencias asociadas con números de línea. La directiva `#line 200` fuerza el número de línea siguiente para que sea 200 (aunque el valor predeterminado es 6) y hasta la siguiente directiva `#line`, el nombre de archivo se notificará como "Especial". La directiva `#line default` devuelve la numeración de líneas a su numeración predeterminada, que cuenta las líneas a las que la directiva anterior ha cambiado el número.

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

La compilación genera el siguiente resultado:

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used
```

Comentarios

La directiva `#line` podría usarse en un paso intermedio automatizado en el proceso de compilación. Por ejemplo, si se han eliminado las líneas del archivo de código fuente original, pero aún quiere que el compilador genere unos resultados en función de la numeración de líneas original en el archivo, puede quitar las líneas y, después, simular la numeración de líneas original con `#line`.

La directiva `#line hidden` oculta las líneas sucesivas del depurador, de forma que, cuando el desarrollador ejecuta paso a paso el código, cualquier línea entre `#line hidden` y la siguiente directiva `#line` (suponiendo que no sea otra directiva `#line hidden`) se depurará paso a paso por procedimientos. Esta opción también se puede usar para permitir que ASP.NET diferencie entre el código generado por el equipo y el definido por el usuario. Aunque ASP.NET es el consumidor principal de esta característica, es probable que la usen más generadores de código fuente.

Una directiva `#line hidden` no afecta a los nombres de archivo o números de línea en el informe de errores. Es decir, si se produce un error en un bloque oculto, el compilador notificará el nombre de archivo y número de línea

reales del error.

La directiva `#line filename` especifica el nombre de archivo que quiere que aparezca en la salida del compilador. De forma predeterminada, se usa el nombre real del archivo de código fuente. El nombre de archivo debe estar entre comillas dobles ("") y debe ir precedido de un número de línea.

Un archivo de código fuente puede tener cualquier número de directivas `#line`.

Ejemplo 1

En el ejemplo siguiente, se muestra cómo el depurador omite las líneas ocultas en el código. Al ejecutar el ejemplo, mostrará tres líneas de texto. En cambio, al establecer un punto de interrupción, como se muestra en el ejemplo, y presionar F10 para recorrer el código, observará que el depurador omite la línea oculta. Tenga también en cuenta que, incluso si configura un punto de interrupción en la línea oculta, el depurador lo omitirá.

```
// preprocessor_linehidden.cs
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine("Normal line #1."); // Set break point here.
#line hidden
        Console.WriteLine("Hidden line.");
#line default
        Console.WriteLine("Normal line #2.");
    }
}
```

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#region (Referencia de C#)

29/06/2020 • 2 minutes to read • Edit Online

`#region` permite especificar un bloque de código que se puede expandir o contraer cuando se usa la característica de esquematización del editor de código. En archivos de código más largos, es conveniente poder contraer u ocultar una o varias regiones para poder centrarse en la parte del archivo en la que se está trabajando actualmente. En el ejemplo siguiente se muestra cómo definir una región:

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Comentarios

Un bloque `#region` se debe terminar con una directiva #endregion.

Un bloque `#region` no se puede superponer con un bloque #if. Pero, un bloque `#region` se puede anidar en un bloque `#if` y un bloque `#if` se puede anidar en un bloque `#region`.

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#endregion (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#endregion` marca el final de un bloque #region. Por ejemplo:

```
#region MyClass definition
class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

#pragma (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#pragma` proporciona al compilador instrucciones especiales para la compilación del archivo en el que aparece. Las instrucciones deben ser compatibles con el compilador. En otras palabras, no puede usar `#pragma` para crear instrucciones de preprocesamiento personalizadas. El compilador de Microsoft C# admite las siguientes dos instrucciones `#pragma`:

#pragma warning

#pragma checksum

Sintaxis

```
#pragma pragma-name pragma-arguments
```

Parámetros

`pragma-name`
El nombre de una pragma reconocida.

`pragma-arguments`
Argumentos específicos de pragma.

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#
- #pragma warning
- #pragma checksum

#pragma warning (Referencia de C#)

18/03/2020 • 2 minutes to read • Edit Online

`#pragma warning` puede habilitar o deshabilitar determinadas advertencias.

Sintaxis

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Parámetros

`warning-list`
Una lista separada por comas de números de advertencia. El prefijo "CS" es opcional.

Cuando no se especifica ningún número de advertencia, `disable` deshabilita todas las advertencias y `restore` habilita todas las advertencias.

> **NOTE**
> Para buscar los números de advertencia en Visual Studio, compile el proyecto y después busque los números de advertencia en la ventana **Salida**.

Ejemplo

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)]  // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

Vea también

- Referencia de C#
- Guía de programación de C#

- Directivas de preprocesador de C#
- Errores del compilador de C#

#pragma checksum (Referencia del programador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Genera sumas de comprobación de archivos de código fuente para ayudar con la depuración de páginas ASP.NET.

Sintaxis

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Parámetros

`"filename"`
El nombre del archivo que requiere la supervisión para cambios o actualizaciones.

`"{guid}"`
El identificador único global (GUID) del algoritmo hash.

`"checksum_bytes"`
La cadena de dígitos hexadecimales que representa los bytes de la suma de comprobación. Debe ser un número par de dígitos hexadecimales. Un número impar de dígitos genera una advertencia de tiempo de compilación y la directiva se ignora.

Comentarios

El depurador de Visual Studio usa una suma de comprobación para asegurarse de que siempre encuentra el código fuente correcto. El compilador calcula la suma de comprobación para un archivo de origen y, después, emite el resultado en el archivo de base de datos del programa (PDB). Después, el depurador usa el archivo PDB para comparar la suma de comprobación que calcula para el archivo de origen.

Esta solución no funciona para proyectos de ASP.NET, ya que la suma de comprobación calculada es para el archivo de código fuente generado, no para el archivo .aspx. Para solucionar este problema, `#pragma checksum` proporciona compatibilidad de la suma de comprobación con páginas ASP.NET.

Cuando se crea un proyecto de ASP.NET en Visual C#, el archivo de código fuente generado contiene una suma de comprobación del archivo .aspx desde el que se genera el código fuente. Después, el compilador escribe esta información en el archivo PDB.

Si el compilador no encuentra ninguna directiva `#pragma checksum` en el archivo, calcula la suma de comprobación y escribe el valor en el archivo PDB.

Ejemplo

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

Vea también

- Referencia de C#
- Guía de programación de C#
- Directivas de preprocesador de C#

Opciones del compilador de C#

18/03/2020 • 2 minutes to read • Edit Online

El compilador genera archivos ejecutables (.exe), archivos de biblioteca de vínculos dinámicos (.dll) o módulos de códigos (.netmodule).

Cada opción del compilador está disponible en dos formatos: **-option** y **/option**. En la documentación solo se muestra el formato **-option**.

En Visual Studio, establecerá las opciones del compilador en el archivo *web.config*. Para más información, vea <compiler> Element (Elemento).

En esta sección

- Compilación de línea de comandos con csc.exe: información sobre la compilación de una aplicación de C# desde la línea de comandos.
- Procedimiento Establecer variables de entorno para la línea de comandos de Visual Studio: se proporcionan los pasos para ejecutar *vsvars32.bat* a fin de habilitar compilaciones de línea de comandos.
- Opciones del compilador de C# enumeradas por categoría: una lista de categorías de las opciones del compilador.
- Opciones del compilador de C# enumeradas alfabéticamente: una lista alfabética de las opciones del compilador.

Secciones relacionadas

- Página Compilar del Diseñador de proyectos: configuración de las propiedades que regulan la compilación y depuración del proyecto. Incluye información sobre los pasos de compilación personalizada en proyectos de Visual C#.
- Compilaciones predeterminadas y personalizadas: información sobre los tipos de compilación y las configuraciones.
- Preparación y administración de compilaciones: procedimientos para compilar en el entorno de desarrollo de Visual Studio.

Compilar la línea de comandos con csc.exe

20/05/2020 • 7 minutes to read • Edit Online

Puede invocar el compilador de C# escribiendo el nombre de su archivo ejecutable (*csc.exe*) en un símbolo del sistema.

Si usa la ventana **Símbolo del sistema para desarrolladores de Visual Studio**, todas las variables de entorno necesarias se establecen automáticamente. Para obtener información sobre cómo acceder a esta herramienta, vea el tema Símbolo del sistema para desarrolladores de Visual Studio.

Si usa una ventana de símbolo del sistema estándar, debe ajustar la ruta de acceso antes de poder invocar *csc.exe* desde cualquier subdirectorio del equipo. También debe ejecutar *vsvars32.bat* para establecer las variables de entorno adecuadas para admitir las compilaciones desde la línea de comandos. Para obtener más información sobre *vsvars32.bat*, incluidas las instrucciones sobre cómo buscarlo y ejecutarlo, vea Procedimiento para establecer variables de entorno para la línea de comandos de Visual Studio.

Si trabaja en un equipo que solo tiene el kit de desarrollo de software (SDK) de Windows, puede usar el compilador de C# en el **símbolo del sistema de SDK**, que puede abrir desde la opción de menú **Microsoft .NET Framework SDK**.

También puede usar MSBuild para compilar programas de C# mediante programación. Para obtener más información, vea MSBuild.

El archivo ejecutable *csc.exe* suele encontrarse en la carpeta Microsoft.NET\Framework\ *<Versión>* del directorio de *Windows*. Su ubicación puede variar, según la configuración exacta de un equipo concreto. Si se instala más de una versión de .NET Framework en el equipo, encontrará varias versiones de este archivo. Para obtener más información sobre estas instalaciones, vea Cómo: Determinar qué versiones de .NET Framework están instaladas.

> **TIP**
>
> Al compilar un proyecto mediante el IDE de Visual Studio, puede mostrar el comando **csc** y las opciones del compilador asociadas en la ventana **Salida**. Para mostrar esta información, siga las instrucciones de Cómo: Ver, guardar y configurar archivos de registro de compilación para cambiar el nivel de detalle de los datos del registro a **Normal** o **Detallado**. Una vez recompilado el proyecto, busque **csc** en la ventana **Salida** para localizar la invocación del compilador de C#.

En este tema

- Reglas para la sintaxis de la línea de comandos
- Líneas de comandos de ejemplo
- Diferencias entre el resultado del compilador de C++ y el compilador de C#

Reglas para la sintaxis de la línea de comandos para el compilador de C#

El compilador de C# utiliza las siguientes reglas para interpretar los argumentos empleados en la línea de comandos del sistema operativo:

- Los argumentos van delimitados por espacio en blanco, que puede ser un carácter de espacio o una tabulación.
- El carácter de intercalación (^) no se reconoce como carácter de escape ni como delimitador. El analizador

de la línea de comandos del sistema operativo procesa este carácter antes de pasarlo a la matriz `argv` del programa.

- Una cadena entre comillas ("cadena") se interpreta como un solo argumento, sin importar el espacio en blanco que contenga. Se puede incrustar una cadena entre comillas dentro de un argumento.
- Unas comillas precedidas por una barra diagonal inversa (\") se interpretan como un literal de cadena de comillas (").
- Las barras diagonales inversas se interpretan literalmente, a menos que precedan inmediatamente a unas comillas.
- Si a un número par de barras diagonales inversas le siguen unas comillas, se coloca una barra diagonal inversa en la matriz `argv` por cada par de barras diagonales y se interpretan las comillas como delimitador de cadenas.
- Si a un número impar de barras diagonales inversas le siguen unas comillas, se coloca una barra diagonal inversa en la matriz `argv` por cada par de barras diagonales y se interpretan las comillas como un carácter de "escape" gracias a la barra diagonal restante. Esto hace que se agregue un literal de comillas dobles (") a `argv`.

Líneas de comandos de ejemplo para el compilador de C#

- Compila *File.cs* y genera *File.exe*:

```
csc File.cs
```

- Compila *File.cs* y genera *File.dll*:

```
csc -target:library File.cs
```

- Compila *File.cs* y crea *My.exe*:

```
csc -out:My.exe File.cs
```

- Compila todos los archivos de C# del directorio actual, con optimizaciones habilitadas y define el símbolo DEBUG. El resultado es *File2.exe*:

```
csc -define:DEBUG -optimize -out:File2.exe *.cs
```

- Compila todos los archivos de C# del directorio actual, generando una versión de depuración de *File2.dll*. No se muestra logotipo ni ningún tipo de advertencias:

```
csc -target:library -out:File2.dll -warn:0 -nologo -debug *.cs
```

- Compila todos los archivos de C# del directorio actual en *Something.xyz* (un DLL):

```
csc -target:library -out:Something.xyz *.cs
```

Diferencias entre el resultado del compilador de C++ y el compilador de C#

No se crean archivos objeto (*.obj*) como resultado de invocar el compilador de C#; se crean directamente archivos de salida. Como consecuencia de ello, el compilador de C# no requiere un vinculador.

Vea también

- Opciones del compilador de C#
- Opciones del compilador de C#, por orden alfabético
- Opciones del compilador de C#, por categoría
- Main() y argumentos de la línea de comandos
- Argumentos de la línea de comandos
- Procedimiento Mostrar argumentos de la línea de comandos
- Valores devueltos de Main()

Establecimiento de variables de entorno para la línea de comandos de Visual Studio

18/03/2020 • 2 minutes to read • Edit Online

El archivo VsDevCmd.bat establece las variables de entorno apropiadas para habilitar compilaciones desde la línea de comandos.

> **NOTE**
> Visual Studio 2015 y versiones anteriores usaban VSVARS32.bat, no VsDevCmd.bat, para el mismo propósito. Este archivo se almacena en \Archivos de programa\Microsoft Visual Studio*Versión*\Common7\Tools o Archivos de programa (x86)\Microsoft Visual Studio*Versión*\Common7\Tools.

Si se instala la versión actual de Visual Studio en un equipo que también tiene una versión anterior de Visual Studio, no se deben ejecutar los archivos VsDevCmd.bat y VSVARS32.BAT de versiones diferentes en la misma ventana del símbolo del sistema. En su lugar, debe ejecutar el comando para cada versión en su propia ventana.

Para ejecutar VsDevCmd.BAT:

1. En el menú **Inicio**, abra el **símbolo del sistema para desarrolladores de VS 2019**. Se encuentra en la carpeta **Visual Studio 2019**.
2. Cambie al subdirectorio \Archivos de programa\Microsoft Visual Studio*Versión**Oferta*\Common7\Tools o Archivos de programa (x86)\Microsoft Visual Studio*Versión**Oferta*\Common7\Tools de la instalación. (*Versión* es *2019* para la versión actual. *Oferta* es una de las opciones siguientes: *Enterprise*, *Professional* o *Community*.)
3. Ejecute VsDevCmd.bat escribiendo **VsDevCmd**.

   > **Caution**
   > VsDevCmd.bat puede ser diferente en equipos distintos. No debe reemplazar un archivo VsDevCmd.bat dañado o inexistente por el archivo VsDevCmd.bat de otro equipo. En su lugar, vuelva a ejecutar el programa de instalación para reemplazar el archivo que falta.

Opciones disponibles para VsDevCmd.BAT

Para ver las opciones disponibles para VsDevCmd.BAT, ejecute el comando con la opción `-help`:

```
VsDevCmd.bat -help
```

Vea también

- Compilar la línea de comandos con csc.exe

Opciones del compilador de C#, por categoría

29/06/2020 • 7 minutes to read • Edit Online

Las opciones del compilador siguientes están ordenadas por categoría. Para acceder a una lista en orden alfabético, vea Opciones del compilador de C#, por orden alfabético.

Optimización

OPCIÓN	PROPÓSITO
-filealign	Especifica el tamaño de las secciones del archivo de salida.
-optimize	Habilita o deshabilita las optimizaciones.

Archivos de resultados

OPCIÓN	PROPÓSITO
-deterministic	Hace que el compilador genere un ensamblado cuyo contenido binario es idéntico en todas las compilaciones si las entradas son idénticas.
-doc	Especifica un archivo XML donde se escribirán comentarios de documentación procesados.
-out	Especifica el archivo de salida.
-pathmap	Especifica una asignación para los nombres de rutas de acceso de origen generados por el compilador
/pdb	Especifica el nombre y la ubicación de archivo del archivo .pdb.
-platform	Especifica la plataforma de salida.
/preferreduilang	Especifica un idioma para los resultados del compilador.
/refout	Genere un ensamblado de referencia además del ensamblado principal.
/refonly	Genere un ensamblado de referencia en lugar del ensamblado principal.
-target	Especifica el formato del archivo de salida con una de estas cinco opciones: -target:appcontainerexe, -target:exe, -target:library, -target:module, -target:winexe o -target:winmdobj.
-modulename:<string>	Especifica el nombre del módulo de origen.

Ensamblados de .NET Framework

OPCIÓN	PROPÓSITO
-addmodule	Especifica uno o varios módulos que formarán parte de este ensamblado.
-delaysign	Indica al compilador que debe agregar la clave pública y mantener el ensamblado sin firma.
-keycontainer	Especifica el nombre del contenedor de claves criptográficas.
-keyfile	Especifica el nombre de archivo que contiene la clave criptográfica.
/lib	Especifica la ubicación de los ensamblados a los que se hace referencia mediante -reference.
-nostdlib	Indica al compilador que no debe importar la biblioteca estándar (mscorlib.dll).
-publicsign	Aplica una clave pública sin firmar el ensamblado, pero establece el bit en el ensamblado lo que indica que el ensamblado está firmado.
-reference	Importa los metadatos de un archivo que contiene un ensamblado.
-analyzer	Ejecuta los analizadores de este ensamblado (forma abreviada: /a).
-additionalfile	Asigna nombre a otros archivos que no afectan directamente a la generación de código, pero que los analizadores pueden usar para generar errores o advertencias.
-embed	Insertar todos los archivos de origen en el archivo PDB
-embed:<file list>	Insertar archivos específicos en el archivo PDB

Comprobación de errores y depuración

OPCIÓN	PROPÓSITO
-bugreport	Crea un archivo que contiene información que permite notificar un error fácilmente.
/checked	Especifica si la aritmética de enteros que desborda los límites del tipo de datos iniciará una excepción en tiempo de ejecución.
-debug	Indica al compilador que debe emitir la información de depuración.
-errorreport	Establece el comportamiento de los informes de errores.

OPCIÓN	PROPÓSITO
/fullpaths	Especifica la ruta de acceso absoluta al archivo en los resultados del compilador.
-nowarn	Suprime la generación de advertencias especificadas en el compilador.
-nullable	Especifica la opción de contexto que admite un valor NULL.
/warn	Establece el nivel de advertencia.
-warnaserror	Promueve las advertencias a errores.
-ruleset:<file>	Especifica un archivo de conjunto de reglas que deshabilita diagnósticos específicos.

Preprocesador

OPCIÓN	PROPÓSITO
-define	Define los símbolos de preprocesador.

Recursos

OPCIÓN	PROPÓSITO
-link	Hace que la información de tipo COM de ensamblados específicos esté disponible en el proyecto.
-linkresource	Crea un vínculo a un recurso administrado.
-resource	Incrusta un recurso de .NET Framework en el archivo de salida.
-win32icon	Especifica un archivo .ico para insertarlo en el archivo de salida.
/win32res:	Especifica un recurso Win32 para insertarlo en el archivo de salida.

Varios

OPCIÓN	PROPÓSITO
@	Especifica un archivo de respuesta.
-?	Enumera las opciones del compilador para stdout.
-baseaddress	Especifica la dirección base preferida para cargar una DLL.

OPCIÓN	PROPÓSITO
-codepage	Especifica la página de códigos que se va a usar para todos los archivos de código fuente de la compilación.
-help	Enumera las opciones del compilador para stdout.
-highentropyva	Especifica que el archivo ejecutable es compatible con la selección aleatoria del diseño del espacio de direcciones (ASLR).
-langversion	Especifique la versión de lenguaje: predeterminada, ISO-1, ISO-2, 3, 4, 5, 6, 7, 7.1, 7.2, 7.3 o más reciente
-main	Especifica la ubicación del método **Main**.
-noconfig	Indica al compilador que no debe compilar con csc.rsp.
-nologo	Suprime la información de titular del compilador.
-recurse	Busca en los subdirectorios los archivos de código fuente que se deben compilar.
-subsystemversion	Especifica la versión mínima del subsistema que el archivo ejecutable puede usar.
/unsafe	Habilita la compilación del código que usa la palabra clave unsafe.
-utf8output	Muestra los resultados del compilador en codificación UTF-8.
-parallel[+|-]	Especifica si hay que usar la compilación simultánea (+).
-checksumalgorithm:<alg>	Especifique el algoritmo para calcular la suma de comprobación del archivo de origen almacenada en el archivo PDB. Los valores admitidos son: SHA256 (valor predeterminado) o SHA1. Debido a problemas de colisión con SHA1, Microsoft recomienda SHA256.

Opciones obsoletas

OPCIÓN	PROPÓSITO
-incremental	Habilita la compilación incremental.

Vea también

- Opciones del compilador de C#
- Opciones del compilador de C#, por orden alfabético
- Establecimiento de variables de entorno para la línea de comandos de Visual Studio

Opciones del compilador de C#, por orden alfabético

29/06/2020 • 8 minutes to read • Edit Online

Las siguientes opciones del compilador están ordenadas alfabéticamente. Para obtener una lista por categoría, vea Opciones del compilador de C# por categoría.

OPCIÓN	PROPÓSITO
@	Leer un archivo de respuesta para ver más opciones.
-?	Muestra un mensaje de uso a stdout.
-additionalfile	Asigna nombre a otros archivos que no afectan directamente a la generación de código, pero que los analizadores pueden usar para generar errores o advertencias.
-addmodule	Vincula los módulos especificados en este ensamblado.
-analyzer	Ejecuta los analizadores de este ensamblado (forma abreviada: -a).
/appconfig	Especifica la ubicación de app.config en el momento del enlace de ensamblado.
-baseaddress	Especifica la dirección base de la biblioteca que se compilará.
-bugreport	Crea un archivo 'Informe de errores'. Este archivo se enviará junto con la información de bloqueo si se usa con -errorreport:prompt o -errorreport:send.
/checked	Hace que el compilador genere comprobaciones de desbordamiento.
-checksumalgorithm:<alg>	Especifica el algoritmo para calcular la suma de comprobación del archivo de origen almacenada en el archivo PDB. Los valores admitidos son: SHA256 (valor predeterminado) o SHA1. Debido a problemas de colisión con SHA1, Microsoft recomienda SHA256.
-codepage	Especifica la página de códigos que se va a utilizar cuando se abren los archivos de código fuente.
-debug	Emite información de depuración.
-define	Define símbolos de compilación condicional.
-delaysign	Retrasa la firma del ensamblado usando solo la parte pública de la clave de nombre seguro.

OPCIÓN	PROPÓSITO
-deterministic	Hace que el compilador genere un ensamblado cuyo contenido binario es idéntico en todas las compilaciones si las entradas son idénticas.
-doc	Especifica un archivo de documentación XML que se generará.
-embed	Insertar todos los archivos de origen en el archivo PDB
-embed:<file list>	Insertar archivos específicos en el archivo PDB
-errorendlocation	Línea y columna de salida de la ubicación final de cada error.
-errorlog:<file>	Especifique un archivo para registrar todos los diagnósticos del compilador y el analizador.
-errorreport	Especifica cómo tratar los errores internos del compilador: prompt, send o none. El valor predeterminado es none.
-filealign	Especifica la alineación usada para las secciones de archivos de salida.
/fullpaths	Hace que el compilador genere rutas de acceso completas.
-help	Muestra un mensaje de uso a stdout.
-highentropyva	Especifica que se admite el ASLR de alta entropía.
-incremental	Habilita la compilación incremental [obsoleto].
-keycontainer	Especifica un contenedor de claves de nombre seguro.
-keyfile	Especifica un archivo de clave de nombre seguro.
-langversion:<string>	Especifique la versión de lenguaje: predeterminada, ISO-1, ISO-2, 3, 4, 5, 6, 7, 7.1, 7.2, 7.3 o más reciente
/lib	Especifica más directorios donde buscar referencias.
-link	Hace que la información de tipo COM de ensamblados específicos esté disponible en el proyecto.
-linkresource	Vincula el recurso especificado a este ensamblado.
-main	Especifica el tipo que contiene el punto de entrada (se pasan por alto los demás puntos de entrada posibles).
-moduleassemblyname	Especifica un ensamblado con tipos no públicos a los que puede acceder un archivo .netmodule.
-modulename:<string>	Especifica el nombre del módulo de origen.

OPCIÓN	PROPÓSITO
-noconfig	Indica al compilador que no incluya el archivo CSC.RSP de forma automática.
-nologo	Suprime el mensaje de copyright del compilador.
-nostdlib	Indica al compilador que no haga referencia a la biblioteca estándar (mscorlib.dll).
-nowarn	Deshabilita mensajes de advertencia específicos.
-nowin32manifest	Indica al compilador que no incruste un manifiesto de la aplicación en el archivo ejecutable.
-nullable	Especifica la opción de contexto que admite un valor NULL.
-optimize	Habilita o deshabilita las optimizaciones.
-out	Especifica el nombre de archivo de salida (nombre predeterminado: nombre base de archivo con la clase principal o del primer archivo).
-parallel[+|-]	Especifica si hay que usar la compilación simultánea (+).
-pathmap	Especifica una asignación para los nombres de rutas de acceso de origen generados por el compilador.
/pdb	Especifica el nombre y la ubicación de archivo del archivo .pdb.
-platform	Limita las plataformas en las que se puede ejecutar este código: x86, Itanium, x64, anycpu o anycpu32bitpreferred. El valor predeterminado es anycpu.
/preferreduilang	Especifica el idioma que se usará para los resultados del compilador.
-publicsign	Aplica una clave pública sin firmar el ensamblado, pero establece el bit en el ensamblado lo que indica que el ensamblado está firmado.
-recurse	Incluye todos los archivos del directorio y subdirectorios actuales de acuerdo con las especificaciones de los comodines.
-reference	Hace referencia a los metadatos de los archivos de ensamblado especificados.
/refout	Genere un ensamblado de referencia además del ensamblado principal.
/refonly	Genere un ensamblado de referencia en lugar del ensamblado principal.

OPCIÓN	PROPÓSITO
-reportanalyzer	Notifica información adicional del analizador, como el tiempo de ejecución.
-resource	Incrusta el recurso especificado.
-ruleset:<file>	Especifica un archivo de conjunto de reglas que deshabilita diagnósticos específicos.
-subsystemversion	Especifica la versión mínima del subsistema que el archivo ejecutable puede usar.
-target	Especifica el formato del archivo de salida con una de las seis opciones siguientes: -target:appcontainerexe, -target:exe, -target:library, -target:module, -target:winexe, -target:winmdobj.
/unsafe	Permite código no seguro.
-utf8output	Genera mensajes del compilador en codificación UTF-8.
-version	Muestra el número de versión del compilador y se cierra.
/warn	Establece el nivel de advertencia (0-4).
-warnaserror	Notifica advertencias específicas como errores.
-win32icon	Usa este icono para los resultados.
-win32manifest	Especifica un archivo de manifiesto win32 personalizado.
/win32res:	Especifica el archivo de recursos win32 (.res).

Vea también

- Opciones del compilador de C#
- Opciones del compilador de C#, por categoría
- Establecimiento de variables de entorno para la línea de comandos de Visual Studio
- Elemento <compiler>

@ (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción @ le permite especificar un archivo que contiene opciones del compilador y archivos de código fuente para compilar.

Sintaxis

```
@response_file
```

Argumentos

`response_file`

Un archivo que enumera las opciones del compilador o los archivos de código fuente que se compilarán.

Comentarios

El compilador procesará las opciones del compilador y los archivos de código fuente como si se hubieran especificado en la línea de comandos.

Para especificar más de un archivo de respuesta en una compilación, especifique varias opciones de archivo de respuesta. Por ejemplo:

```
@file1.rsp @file2.rsp
```

En un archivo de respuesta, puede haber varias opciones del compilador y archivos de código fuente en una sola línea. Una sola especificación de opción del compilador debe aparecer en una línea (no puede abarcar varias líneas). Los archivos de respuesta pueden tener comentarios que comiencen con el símbolo #.

Especificar opciones del compilador desde un archivo de respuesta es igual que enviar esos comandos en la línea de comandos. Vea Compilar desde la línea de comandos para obtener más información.

El compilador procesa las opciones de comando a medida que se encuentran. Por tanto, los argumentos de línea de comandos pueden reemplazar las opciones enumeradas anteriormente en archivos de respuesta. Por el contrario, las opciones en un archivo de respuesta reemplazarán las opciones enumeradas anteriormente en la línea de comandos o en otros archivos de respuesta.

C# proporciona el archivo csc.rsp, que se encuentra en el mismo directorio que el archivo csc.exe. Vea -noconfig para saber más sobre csc.rsp.

No se puede establecer esta opción del compilador en el entorno de desarrollo de Visual Studio, ni se puede cambiar mediante programación.

Ejemplo

Estas son algunas líneas de un archivo de respuesta de ejemplo:

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

Vea también

- Opciones del compilador de C#

-addmodule (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Esta opción agrega un módulo que se ha creado con el modificador target:module para la compilación actual.

Sintaxis

```
-addmodule:file[;file2]
```

Argumentos

`file`, `file2`
Un archivo de salida que contiene metadatos. El archivo no puede contener un manifiesto de ensamblado. Para importar más de un archivo, hay que separar los nombres de archivo con una coma o un punto y coma.

Comentarios

Todos los módulos agregados mediante **-addmodule** deben encontrarse en el mismo directorio que el archivo de salida en tiempo de ejecución. Es decir, puede especificar un módulo de cualquier directorio en el momento de la compilación, pero el módulo debe encontrarse en el directorio de la aplicación en tiempo de ejecución. Si dicho módulo no se encuentra en el directorio de la aplicación en tiempo de ejecución, obtendrá TypeLoadException.

`file` no puede contener ningún ensamblado. Por ejemplo, si el archivo de salida se ha creado con -target:module, se pueden importar sus metadatos con **-addmodule**.

Si se ha creado el archivo de salida con una opción **-target** diferente de **-target:module**, no se podrán importar sus metadatos con **-addmodule**, pero sí con -reference.

Esta opción del compilador no está disponible en Visual Studio; en un proyecto no se puede hacer referencia a un módulo. Además, esta opción del compilador no se puede modificar mediante programación.

Ejemplo

Compile el archivo de código fuente `input.cs` y agregue metadatos de `metad1.netmodule` y `metad2.netmodule` para generar `out.exe`:

```
csc -addmodule:metad1.netmodule;metad2.netmodule -out:out.exe input.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones
- Ensamblados de múltiples archivos
- Compilar un ensamblado de varios archivos

-appconfig (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

La opción del compilador **-appconfig** permite a una aplicación de C# especificar la ubicación del archivo de configuración de la aplicación de un ensamblado (app.config) al Common Language Runtime (CLR) en tiempo de enlace del ensamblado.

Sintaxis

```
-appconfig:file
```

Argumentos

`file`

Obligatorio. El archivo de configuración de la aplicación que contiene los valores de enlace del ensamblado.

Comentarios

Un uso de **-appconfig** es el de escenarios avanzados en los que un ensamblado tiene que hacer referencia a la versión de .NET Framework y a la versión de .NET Framework para Silverlight de un ensamblado de referencia determinado al mismo tiempo. Por ejemplo, un diseñador de XAML escrito en Windows Presentation Foundation (WPF) podría tener que hacer referencia al escritorio de WPF, para la interfaz de usuario del diseñador, y al subconjunto de WPF que se incluye con Silverlight. El mismo ensamblado del diseñador tiene que tener acceso a ambos ensamblados. De forma predeterminada, las referencias independientes producen un error del compilador, ya que el enlace del ensamblado considera los dos ensamblados equivalentes.

La opción del compilador **-appconfig** permite especificar la ubicación de un archivo app.config que deshabilita el comportamiento predeterminado usando una etiqueta `<supportPortability>`, como se muestra en el siguiente ejemplo.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

El compilador pasa la ubicación del archivo a la lógica de enlace del ensamblado de CLR.

> **NOTE**
>
> Si está usando Microsoft Build Engine (MSBuild) para compilar su aplicación, puede establecer la opción del compilador **-appconfig** mediante la adición de una etiqueta de propiedad al archivo .csproj. Para usar el archivo app.config que ya está establecido en el proyecto, agregue la etiqueta `<UseAppConfigForCompiler>` de propiedades al archivo .csproj y establezca su valor en `true`. Para especificar otro archivo app.config, agregue la etiqueta `<AppConfigForCompiler>` de propiedades y establezca su valor en la ubicación del archivo.

Ejemplo

En el siguiente ejemplo se muestra un archivo app.config que permite a una aplicación tener referencias a la implementación de .NET Framework y de .NET Framework para Silverlight de cualquier ensamblado de .NET Framework que exista en ambas implementaciones. La opción del compilador **-appconfig** especifica la ubicación de este archivo app.config.

```
<configuration>
      <runtime>
      <assemblyBinding>
            <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
            <supportPortability PKT="31bf3856ad364e35" enable="false"/>
      </assemblyBinding>
      </runtime>
</configuration>
```

Vea también

- <supportPortability> Element (Elemento)
- Opciones del compilador de C#, por orden alfabético

-baseaddress (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-baseaddress** permite especificar la dirección base preferida para cargar un archivo DLL. Para obtener más información sobre cuándo y por qué usar esta opción, vea el blog de Larry Osterman.

Sintaxis

```
-baseaddress:address
```

Argumentos

`address`

La dirección base del archivo DLL. Esta dirección puede especificarse como un número octal, hexadecimal o decimal.

Comentarios

La dirección base predeterminada para un archivo DLL se establece mediante Common Language Runtime de .NET Framework.

Tenga en cuenta que la palabra de orden inferior en esta dirección se redondeará. Por ejemplo, si especifica 0x11110001, se redondeará a 0x11110000.

Para completar el proceso de firma para un archivo DLL, use SN.EXE con la opción -R.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar**.
3. Haga clic en el botón **Avanzada** .
4. Modifique la propiedad **Dirección base del archivo DLL**.

   Para establecer esta opción del compilador mediante programación, vea BaseAddress.

Vea también

- ProcessModule.BaseAddress
- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-bugreport (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Especifica que esa información de depuración debe colocarse en un archivo para su análisis posterior.

Sintaxis

```
-bugreport:file
```

Argumentos

`file`
El nombre del archivo que quiere que contenga su informe de errores.

Comentarios

La opción **-bugreport** especifica que la siguiente información debe colocarse en `file`:

- Una copia de todos los archivos de código fuente de la compilación.
- Una lista de las opciones del compilador que se han usado en la compilación.
- Información de la versión sobre su compilador, tiempo de ejecución y sistema operativo.
- Módulos y ensamblados a los que se hace referencia, guardados como dígitos hexadecimales, excepto los ensamblados que se proporcionan con .NET Framework y SDK.
- Resultados del compilador, si los hay.
- Una descripción del problema, que se le pedirá.
- Una descripción de cómo cree que debe corregirse el problema, que se le pedirá.

Si esta opción se usa con **-errorreport:prompt** o **-errorreport:send**, la información del archivo se enviará a Microsoft Corporation.

Como una copia de todos los archivos de código fuente se colocarán en `file`, puede que quiera reproducir el defecto en el código sospechoso en el programa más corto posible.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Tenga en cuenta que el contenido del código fuente expuesto del archivo generado puede provocar la divulgación de información involuntaria.

Vea también

- Opciones del compilador de C#
- -errorreport (Opciones del compilador de C#)
- Administrar propiedades de proyectos y de soluciones

-checked (Opciones del compilador de C#)

19/03/2020 • 3 minutes to read • Edit Online

La opción **-checked** especifica si una instrucción aritmética de enteros que produce un valor fuera del intervalo del tipo de datos y que no está en el ámbito de las palabras clave checked o unchecked provocará una excepción en tiempo de ejecución.

Sintaxis

```
-checked[+ | -]
```

Comentarios

Una instrucción aritmética de enteros que está en el ámbito de la palabra clave `checked` o `unchecked` no está sujeta al efecto de la opción **-checked**.

Si una instrucción aritmética de enteros que no está en el ámbito de la palabra clave `checked` o `unchecked` produce un valor fuera del intervalo del tipo de datos, y si se usa **-checked+** (o **-checked**) en la compilación, la instrucción provoca una excepción en tiempo de ejecución. Si se usa **-checked-** en la compilación, la instrucción no produce una excepción en tiempo de ejecución.

El valor predeterminado para esta opción es **-checked-** y la comprobación de desbordamiento está deshabilitada.

A veces, las herramientas automatizadas que se usan para compilar aplicaciones de gran tamaño establecen -check en +. Una situación para usar -checked- es reemplazar el valor predeterminado global de la herramienta especificando -checked-.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto. Para obtener más información, consulte Compilar (Página, Diseñador de proyectos) (C#).
2. Haga clic en la página de propiedades de **Compilar**.
3. Haga clic en el botón **Avanzada** .
4. Modifique la propiedad **Comprobar el desbordamiento aritmético**.

Para acceder a esta opción del compilador mediante programación, vea CheckForOverflowUnderflow.

Ejemplo

El comando siguiente compila `t2.cs`. El uso de `-checked` en el comando especifica que cualquier instrucción de aritmética de enteros del archivo que no esté en el ámbito de la palabra clave `checked` o `unchecked` y que produzca un valor fuera del intervalo del tipo de datos provocará una excepción en tiempo de ejecución.

```
csc t2.cs -checked
```

Vea también

- Opciones del compilador de C#

- Administrar propiedades de soluciones y proyectos

-codepage (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Esta opción especifica qué página de códigos se va a usar durante la compilación si la página necesaria no es la página de códigos predeterminada actual del sistema.

Sintaxis

```
-codepage:id
```

Argumentos

`id`

El id. de la página de códigos que se va a usar para todos los archivos de código fuente de la compilación.

Comentarios

El compilador primero intenta interpretar todos los archivos de código fuente como UTF-8. Si los archivos de código fuente se encuentran en una codificación distinta de UTF-8 y usan caracteres que no sean ASCII de 7 bits, emplee la opción **-codepage** para especificar qué página de códigos debe usarse. **-codepage** se aplica a todos los archivos de código fuente de su compilación.

Vea GetCPInfo para obtener información sobre cómo buscar las páginas de códigos que se admiten en su sistema.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-debug (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

La opción **-debug** da lugar a que el compilador genere información de depuración y la incluya en el archivo o los archivos de salida.

Sintaxis

```
-debug[+ | -]
-debug:{full | pdbonly}
```

Argumentos

`+` | `-`
Si se especifica `+`, o simplemente **-debug**, el compilador genera información de depuración y la incluye en una base de datos de programa (archivo .pdb). Si se especifica `-`, que es la opción predeterminada si no se especifica **-debug**, no se crea información de depuración.

`full` | `pdbonly`
Especifica el tipo de información de depuración generado por el compilador. El argumento full, que es la opción predeterminada si no se especifica **-debug:pdbonly**, permite asociar un depurador al programa que se ejecuta. Al especificar la opción pdbonly, se permite depurar el código fuente cuando se inicia el programa en el depurador, pero solo muestra el ensamblador cuando el programa que se ejecuta está asociado al depurador.

Comentarios

Use esta opción para crear versiones de depuración. Si no se especifica **-debug**, **-debug+** o **-debug:full**, no podrá depurar el archivo de salida del programa.

Si usa **-debug:full**, sea consciente de que se producirá algún impacto en la velocidad y el tamaño del código optimizado JIT y un pequeño impacto en la calidad del código con **-debug:full**. Recomendamos **-debug:pdbonly** o ningún PDB para generar el código de la versión de lanzamiento.

> **NOTE**
> Una diferencia entre **-debug:pdbonly** y **-debug:full** es que con **-debug:full** el compilador emite DebuggableAttribute, que se usa para indicar al compilador JIT que la información de depuración está disponible. Por consiguiente, aparecerá un error si el código contiene DebuggableAttribute establecido en false cuando se usa **-debug:full**.

Para obtener más información sobre cómo configurar el rendimiento de depuración de una aplicación, vea Facilitar la depuración de una imagen.

Para cambiar la ubicación del archivo .pdb, vea -pdb (Opciones del compilador de C#).

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar**.
3. Haga clic en el botón **Avanzada** .

4. Modifique la propiedad **Información de depuración**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea DebugSymbols.

Ejemplo

Incluir información de depuración en el archivo de salida `app.pdb`:

```
csc -debug -pdb:app.pdb test.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-define (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

La opción **-define** define `name` como un símbolo en todos los archivos de código fuente de su programa.

Sintaxis

```
-define:name[;name2]
```

Argumentos

`name`, `name2`
El nombre de uno o más símbolos que quiere definir.

Comentarios

La opción **-define** provoca el mismo efecto que usar una directiva del preprocesador #define, salvo que la opción del compilador está en vigor para todos los archivos del proyecto. Un símbolo permanece definido en un archivo de origen hasta que una directiva #undef en el archivo de origen quita la definición. Cuando usa la opción -define, una directiva `#undef` en un archivo no tiene ningún efecto en otros archivos de código fuente del proyecto.

Los símbolos creados por esta opción se pueden usar con #if, #else, #elif y #endif para compilar los archivos de origen condicionalmente.

-d es la forma abreviada de **-define**.

Se pueden definir varios símbolos con **-define** mediante un punto y coma o una coma para separar los nombres de símbolos. Por ejemplo:

```
-define:DEBUG;TUESDAY
```

El propio compilador de C# no define ningún símbolo o macro que puede usar en su código fuente; todas las definiciones de símbolo deben definirse por el usuario.

> **NOTE**
> El valor `#define` de C# no permite que se le proporcione un valor a un símbolo; como en lenguajes como C++. Por ejemplo, `#define` no puede usarse para crear una macro o para definir una constante. Si necesita definir una constante, use una variable `enum`. Si quiere crear una macro de estilo de C++, considere alternativas como genéricos. Como las macros son notoriamente propensas a errores, C# deshabilita su uso pero proporciona alternativas más seguras.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. En la pestaña **Compilar**, escriba el símbolo que va a definirse en el cuadro **Símbolos de compilación condicional**. Por ejemplo, si está usando el ejemplo de código siguiente, simplemente escriba `xx` en el cuadro de texto.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea DefineConstants.

Ejemplo

```
// preprocessor_define.cs
// compile with: -define:xx
// or uncomment the next line
// #define xx
using System;
public class Test
{
    public static void Main()
    {
        #if (xx)
            Console.WriteLine("xx defined");
        #else
            Console.WriteLine("xx not defined");
        #endif
    }
}
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-delaysign (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Esta opción hace que el compilador reserve espacio en el archivo de salida de manera que se pueda agregar una firma digital más adelante.

Sintaxis

```
-delaysign[ + | - ]
```

Argumentos

`+` | `-`

Use **-delaysign-** para firmar completamente un ensamblado. Use **-delaysign+** si quiere incluir solo la clave pública en el ensamblado. El valor predeterminado es **-delaysign-** .

Comentarios

La opción **-delaysign** no tiene ningún efecto a menos que se use con -keyfile o -keycontainer.

Las opciones **-delaysign** y **-publicsign** son mutuamente excluyentes.

Cuando se solicita un ensamblado totalmente firmado, el compilador genera un valor hash para el archivo que contiene el manifiesto (metadatos del ensamblado) y firma dicho valor mediante la clave privada. Esta operación crea una firma digital que se almacena en el archivo que contiene el manifiesto. Cuando se retrasa la firma de un ensamblado, el compilador no calcula ni almacena la firma, sino que reserva espacio en el archivo para que la firma se pueda agregar más tarde.

Por ejemplo, si se usa **-delaysign+** , los evaluadores podrán colocar el ensamblado en la memoria caché global. Después de realizar las pruebas, coloque la clave privada en el ensamblado mediante la utilidad Assembly Linker para firmar el ensamblado por completo.

Para obtener más información, vea Crear y usar ensamblados con nombre seguro y Delay Signing an Assembly (Retrasar la firma de un ensamblado).

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Modifique la propiedad **Solo retrasar firma**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea DelaySign.

Vea también

- Opción C# -publicsign
- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-deterministic

18/03/2020 • 3 minutes to read • Edit Online

Hace que el compilador genere un ensamblado cuya salida byte a byte es idéntica en todas las compilaciones para las entradas idénticas.

Sintaxis

```
-deterministic
```

Comentarios

De forma predeterminada, la salida del compilador para un conjunto determinado de entradas es única, ya que el compilador agrega una marca de tiempo y un GUID que se genera a partir de números aleatorios. Use la opción `-deterministic` para generar un *ensamblado determinista*, cuyo contenido binario es idéntico en todas las compilaciones, siempre y cuando la entrada siga siendo la misma.

El compilador tiene en cuenta las entradas siguientes con el fin de garantizar el determinismo:

- La secuencia de parámetros de la línea de comandos.
- El contenido del archivo de respuesta .rsp del compilador.
- La versión exacta del compilador que se usa y los ensamblados a los que se hace referencia.
- La ruta de acceso del directorio actual.
- El contenido binario de todos los archivos pasados explícitamente al compilador, ya sea de manera directa o indirecta, incluidos los siguientes:
  - Archivos de código fuente
  - Ensamblados a los que se hace referencia
  - Módulos a los que se hace referencia
  - Recursos
  - Archivo de clave de nombre seguro
  - Archivos de respuesta @
  - Analizadores
  - Conjuntos de reglas
  - Archivos adicionales que podrían usar los analizadores
- La referencia cultural actual (para el idioma en el que se producen los diagnósticos y los mensajes de excepción).
- La codificación predeterminada (o página de códigos actual) si no se especifica la codificación.
- La existencia (o la inexistencia) de archivos y su contenido en las rutas de búsqueda del compilador (especificada, por ejemplo, mediante `-lib` o `-recurse`).
- La plataforma CLR en la que se ejecuta el compilador.
- El valor de `%LIBPATH%`, que pueden afectar a la carga de dependencias del analizador.

Cuando los orígenes están disponibles públicamente, se puede usar la compilación determinista para establecer si un archivo binario se compila a partir de una fuente de confianza. También puede ser útil en un sistema de compilación continua para determinar si es necesario ejecutar pasos de compilación que dependen de los cambios realizados en un archivo binario.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-doc (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

La opción **-doc** permite insertar comentarios de documentación en un archivo XML.

Sintaxis

```
-doc:file
```

Argumentos

`file`
Archivo de salida para XML, que se llena con los comentarios de los archivos de código fuente de la compilación.

Comentarios

En los archivos de código fuente, es posible procesar y agregar al archivo XML los comentarios de documentación que preceden a lo siguiente:

- Tipos definidos por el usuario como una clase, un delegado o una interfaz
- Miembros como un campo, un evento, una propiedad o un método

El archivo de código fuente que contiene Main es la primera salida del XML.

Para usar el archivo .xml generado con la función IntelliSense, deje que el nombre del archivo .xml sea igual que el del ensamblado que quiere admitir y, después, asegúrese de que el archivo .xml se encuentra en el mismo directorio que el ensamblado. De este modo, cuando se hace referencia al ensamblado en el proyecto de Visual Studio, también se encuentra el archivo .xml. Vea Proporcionar comentarios del código para obtener más información.

A menos que compile con -target:module, `file` contendrá etiquetas <assembly></assembly> que especifican el nombre del archivo que contiene el manifiesto del ensamblado para el archivo de salida de la compilación.

> **NOTE**
> La opción -doc se aplica a todos los archivos de entrada o, si se establece en la configuración del proyecto, a todos los archivos del proyecto. Para deshabilitar las advertencias relacionadas con los comentarios de documentación para un archivo específico o una sección de código, use #pragma warning.

Vea Etiquetas recomendadas para comentarios de documentación para conocer las maneras de generar documentación a partir de comentarios en el código.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la pestaña **Compilar**.
3. Modifique la propiedad **Archivo de documentación XML**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea DocumentationFile.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-errorreport (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

Esta opción proporciona una forma cómoda de notificar un error interno del compilador de C# a Microsoft.

> **NOTE**
> En Windows Vista y Windows Server 2008, la configuración de informes de errores para Visual Studio no reemplaza la configuración realizada a través de Informe de errores de Windows (WER). La configuración de WER siempre tiene prioridad sobre la configuración de informes de errores de Visual Studio.

Sintaxis

```
-errorreport:{ none | prompt | queue | send }
```

Argumentos

none
No se recopilarán informes sobre errores internos del compilador ni se enviarán a Microsoft.

prompt: pregunta si desea enviar un informe cuando recibe un error interno del compilador. **prompt** es el valor predeterminado cuando se compila una aplicación en el entorno de desarrollo.

queue: pone en cola el informe de error. Cuando inicia sesión con credenciales administrativas, puede notificar los errores que se han producido desde la última vez que ha iniciado sesión. No se le solicitará que envíe informes de error más de una vez cada tres días. **queue** es el valor predeterminado cuando se compila una aplicación en la línea de comandos.

send: envía automáticamente informes de errores internos del compilador a Microsoft. Para habilitar esta opción, debe aceptar la directiva de recopilación de datos de Microsoft. La primera vez que especifique **-errorreport:send** en un equipo, un mensaje del compilador le remitirá a un sitio web que contiene la directiva de recopilación de datos de Microsoft.

Comentarios

Cuando el compilador no puede procesar un archivo de código fuente, se produce un error interno del compilador (ICE). Cuando se produce un ICE, el compilador no genera un archivo de salida ni otro tipo de diagnóstico útil que pueda usar para corregir el código.

En versiones anteriores, cuando recibía un ICE, se le recomendaba que se pusiera en contacto con los servicios de soporte técnico de Microsoft para informar del problema. Mediante el uso de **-errorreport**, puede proporcionar información sobre los ICE al equipo de Visual C#. Sus informes de error pueden ayudar a mejorar las futuras versiones del compilador.

La capacidad de un usuario de enviar informes depende de los permisos de directiva de equipo y de usuario.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto. Para obtener más información, consulte Compilar (Página, Diseñador de proyectos) (C#).

2. Haga clic en la página de propiedades de **Compilar**.

3. Haga clic en el botón **Avanzada** .

4. Modifique la propiedad **Informe de errores internos del compilador**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea ErrorReport.

Vea también

- Opciones del compilador de C#

-filealign (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-filealign** permite especificar el tamaño de las secciones en el archivo de salida.

Sintaxis

```
-filealign:number
```

Argumentos

`number`

Un valor que especifica el tamaño de las secciones del archivo de salida. Los valores válidos son 512, 1024, 2048, 4096 y 8192. Estos valores están en bytes.

Comentarios

Cada sección se alineará en un límite que es un múltiplo del valor **-filealign**. No hay ningún valor predeterminado fijo. Si no se especifica **-filealign**, Common Language Runtime elige un valor predeterminado en tiempo de compilación.

Al especificar el tamaño de la sección, el tamaño del archivo de salida se ve afectado. Modificar el tamaño de la sección puede ser útil para los programas que se ejecutan en dispositivos más pequeños.

Use DUMPBIN para ver información sobre las secciones en el archivo de salida.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar**.
3. Haga clic en el botón **Avanzada** .
4. Modifique la propiedad **Alineación de archivo**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea FileAlignment.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-fullpaths (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-fullpaths** hace que el compilador especifique una ruta de acceso completa al archivo cuando se muestran los errores de compilación y las advertencias.

Sintaxis

```
-fullpaths
```

Comentarios

De manera predeterminada, los errores y advertencias que se producen de la compilación especifican el nombre del archivo en el que se ha detectado el error. La opción **-fullpaths** hace que el compilador especifique una ruta de acceso completa al archivo.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Vea también

- Opciones del compilador de C#

-help, -? (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Esta opción envía un listado de opciones del compilador, y una breve descripción de cada opción, para stdout.

Sintaxis

```
-help
-?
```

Comentarios

Si esta opción se incluye en una compilación, no se creará ningún archivo de salida y no tendrá lugar ninguna compilación.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-highentropyva (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción del compilador **-highentropyva** indica al kernel de Windows si un archivo ejecutable determinado admite la selección aleatoria del diseño del espacio de direcciones (ASLR) de alta entropía.

Sintaxis

```
-highentropyva[+ | -]
```

Argumentos

`+` | `-`

Esta opción especifica que un archivo ejecutable de 64 bits o un archivo ejecutable que está marcado con la opción del compilador -platform:anycpu admite un espacio de direcciones virtuales de alta entropía. La opción está deshabilitada de forma predeterminada. Para habilitarla, use **-highentropyva+** o **-highentropyva**.

Comentarios

La opción **-highentropyva** habilita las versiones compatibles del kernel de Windows para usar niveles superiores de entropía al aleatorizar el diseño del espacio de direcciones de un proceso como parte de ASLR. El uso de niveles superiores de entropía significa que un mayor número de direcciones se puede asignar a las regiones de memoria como pilas y montones. Como resultado, la ubicación de un área de memoria específica es más difícil de adivinar.

Cuando se especifica la opción del compilador **-highentropyva**, el archivo ejecutable de destino y todos los módulos de los que depende deben ser capaces de controlar los valores de puntero que son superiores a 4 gigas (GB), cuando se ejecutan como un proceso de 64 bits.

-keycontainer (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Especifica el nombre del contenedor de claves criptográficas.

Sintaxis

```
-keycontainer:string
```

Argumentos

`string`
El nombre del contenedor de claves de nombre seguro.

Comentarios

Cuando se utiliza la opción **-keycontainer**, el compilador crea un componente que se puede compartir. El compilador inserta una clave pública del contenedor especificado en el manifiesto del ensamblado y firma el último ensamblado con la clave privada. Para generar un archivo de claves, escriba `sn -k file` en la línea de comandos. `sn -i` instala el par de claves en un contenedor. Esta opción no se admite cuando el compilador se ejecuta en CoreCLR. Para firmar un ensamblado al compilar en CoreCLR, utilice la opción -keyfile.

Si se compila con -target:module, el nombre del archivo de claves se mantiene en el módulo y se incorpora al ensamblado al compilar este módulo en un ensamblado con -addmodule.

También se puede especificar esta opción como un atributo personalizado (System.Reflection.AssemblyKeyNameAttribute) en el código fuente de cualquier módulo del Lenguaje Intermedio de Microsoft (MSIL).

También se puede pasar la información de cifrado al compilador con -keyfile. Use -delaysign si quiere agregar la clave pública al manifiesto del ensamblado pero retrasar la firma del ensamblado hasta que se haya probado.

Para obtener más información, vea Crear y usar ensamblados con nombre seguro y Delay Signing an Assembly (Retrasar la firma de un ensamblado).

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Esta opción del compilador no está disponible en el entorno de desarrollo de Visual Studio.

Puede tener acceso mediante programación a esta opción del compilador con AssemblyKeyContainerName.

Vea también

- Opción -keyfile del compilador de C#
- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-keyfile (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

Especifica el nombre de archivo que contiene la clave criptográfica.

Sintaxis

```
-keyfile:file
```

Argumentos

TÉRMINO	DEFINICIÓN
`file`	El nombre del archivo que contiene la clave de nombre seguro.

Comentarios

Cuando se usa esta opción, el compilador inserta la clave pública del archivo especificado en el manifiesto del ensamblado y, después, firma el último ensamblado con la clave privada. Para generar un archivo de clave, escriba sn -k `file` en la línea de comandos.

Si se compila con la opción **-target:module**, el nombre del archivo de claves se mantiene en el módulo y se incorpora en el ensamblado que se crea al compilar un ensamblado con la opción -addmodule.

También puede pasar la información de cifrado al compilador con -keycontainer. Use -delaysign para firmar el ensamblado de forma parcial.

Si se especifica tanto -keyfile como -keycontainer (ya sea mediante una opción de línea de comandos o mediante un atributo personalizado) en la misma compilación, el compilador probará primero el contenedor de claves. Si lo consigue, el ensamblado se firma con la información del contenedor de claves. Si el compilador no encuentra el contenedor de claves, probará el archivo especificado con -keyfile. Si lo consigue, el ensamblado se firma con la información del archivo de clave y la información de la clave se instalará en el contenedor de claves (similar a sn -i) de modo que, en la próxima compilación, el contenedor de claves será válido.

Tenga en cuenta que un archivo de clave puede contener solo la clave pública.

Para obtener más información, vea Crear y usar ensamblados con nombre seguro y Delay Signing an Assembly (Retrasar la firma de un ensamblado).

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades **Firma**.
3. Modifique la propiedad **Seleccione un archivo de clave de nombre seguro**.

Puede tener acceso mediante programación a esta opción del compilador con AssemblyOriginatorKeyFile.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-langversion (Opciones del compilador de C#)

29/06/2020 • 7 minutes to read • Edit Online

Hace que el compilador acepte solo la sintaxis que se incluye en la especificación elegida del lenguaje C#.

Sintaxis

```
-langversion:option
```

Argumentos

`option`

Valores válidos son:

VALOR	SIGNIFICADO
`preview`	El compilador acepta toda la sintaxis de lenguaje válida de la última versión preliminar.
`latest`	El compilador acepta la sintaxis de la última versión del compilador (incluida las versión secundaria).
`latestMajor` (`default`)	El compilador acepta la sintaxis de la versión principal más reciente del compilador.
`8.0`	El compilador acepta solo la sintaxis que se incluye en C# 8.0 o versiones anteriores.
`7.3`	El compilador acepta solo la sintaxis que se incluye en C# 7.3 o versiones anteriores.
`7.2`	El compilador acepta solo la sintaxis que se incluye en C# 7.2 o versiones anteriores.
`7.1`	El compilador acepta solo la sintaxis que se incluye en C# 7.1 o versiones anteriores.
`7`	El compilador acepta solo la sintaxis que se incluye en C# 7.0 o versiones anteriores.
`6`	El compilador acepta solo la sintaxis que se incluye en C# 6.0 o versiones anteriores.
`5`	El compilador acepta solo la sintaxis que se incluye en C# 5.0 o versiones anteriores.
`4`	El compilador acepta solo la sintaxis que se incluye en C# 4.0 o versiones anteriores.

VALOR	SIGNIFICADO
`3`	El compilador acepta solo la sintaxis que se incluye en C# 3.0 o versiones anteriores.
`ISO-2` (o `2`)	El compilador acepta solo la sintaxis que se incluye en ISO/IEC 23270:2006 C# (2.0).
`ISO-1` (o `1`)	El compilador acepta solo la sintaxis que se incluye en ISO/IEC 23270:2003 C# (1.0/1.2).

La versión de idioma predeterminada depende de la plataforma de destino de la aplicación y la versión del SDK o de Visual Studio instalada. Estas reglas se definen en el artículo sobre la configuración de la versión del lenguaje.

Comentarios

Los metadatos a los que hace referencia la aplicación de C# no están sujetos a la opción del compilador **-langversion**.

Dado que cada versión del compilador de C# contiene las extensiones para la especificación del lenguaje, **-langversion** no ofrece las funciones equivalentes de una versión anterior del compilador.

Además, aunque las actualizaciones de versión de C# generalmente coinciden con las versiones de .NET Framework principales, la sintaxis y las características nuevas no están necesariamente asociadas a esa versión de marco específica. Aunque las nuevas características necesitan definitivamente una nueva actualización del compilador que también se publica junto con la revisión de C#, cada característica específica tiene su propia API mínima de .NET o requisitos de Common Language Runtime que pueden permitir que se ejecute en marcos de versiones anteriores mediante la inclusión de paquetes NuGet u otras bibliotecas.

Independientemente de la configuración de **-langversion** que use, usa la versión actual de Common Language Runtime para crear el archivo .exe o .dll. Una excepción son los ensamblados de confianza y -moduleassemblyname (Opción del compilador de C#), que funcionan en **-langversion:ISO-1**.

Para obtener otras formas de especificar la versión del lenguaje C#, consulte el artículo Selección de la versión del lenguaje C#.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea LanguageVersion.

Especificación del lenguaje C#

VERSIÓN	LINK	DESCRIPCIÓN
C# 7.0 y posterior		No está disponible actualmente
C# 6.0	Vínculo	Versión 6 de la especificación del lenguaje C#, borrador no oficial: .NET Foundation
C# 5.0	Descargar PDF	Norma ECMA-334 estándar, quinta edición
C# 3.0	Decargar DOC	Especificación del lenguaje C#, versión 3.0: Microsoft Corporation

VERSIÓN	LINK	DESCRIPCIÓN
C# 2.0	Descargar PDF	Norma ECMA-334 estándar, cuarta edición
C# 1.2	Decargar DOC	Especificación del lenguaje C#, versión 1.2: Microsoft Corporation
C# 1.0	Decargar DOC	Especificación del lenguaje C#, versión 1.0: Microsoft Corporation

Versión del SDK mínima necesaria para admitir todas las características del lenguaje

En la tabla siguiente se enumeran las versiones mínimas del SDK con el compilador de C# que admite la versión del lenguaje correspondiente:

VERSIÓN DE C#	VERSIÓN MÍNIMA DEL SDK
C# 8.0	Microsoft Visual Studio/Build Tools 2019, versión 16.3 o SDK de .NET Core 3.0
C# 7.3	Microsoft Visual Studio/Build Tools 2017, versión 15.7
C# 7.2	Microsoft Visual Studio/Build Tools 2017, versión 15.5
C# 7.1	Microsoft Visual Studio/Build Tools 2017, versión 15.3
C# 7.0	Microsoft Visual Studio/Build Tools 2017
C# 6	Microsoft Visual Studio/Build Tools 2015
C# 5	Microsoft Visual Studio/Build Tools 2012 o compilador empaquetado de .NET Framework 4.5
C# 4	Microsoft Visual Studio/Build Tools 2010 o compilador empaquetado de .NET Framework 4.0
C# 3	Microsoft Visual Studio/Build Tools 2008 o compilador empaquetado de .NET Framework 3.5
C# 2	Microsoft Visual Studio/Build Tools 2005 o compilador empaquetado de .NET Framework 2.0
C# 1.0/1.2	Microsoft Visual Studio/Build Tools .NET 2002 o compilador empaquetado de .NET Framework 1.0

Vea también

- Opciones del compilador de C#
- Administrar propiedades de soluciones y proyectos

-lib (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

La opción **-lib** especifica la ubicación de los ensamblados a los que se hace referencia mediante la opción -reference (Opciones del compilador de C#).

Sintaxis

```
-lib:dir1[,dir2]
```

Argumentos

`dir1`

Un directorio para el compilador para buscar un ensamblado al que se hace referencia si no lo encuentra en el directorio de trabajo actual (el directorio desde el que se invoca al compilador) o en el directorio del sistema de Common Language Runtime.

`dir2`

Uno o varios directorios adicionales para buscar las referencias a ensamblados. Separe los nombres de directorio adicionales con una coma y sin espacio en blanco entre ellos.

Comentarios

El compilador busca referencias a ensamblados que no presentan la ruta completa en el siguiente orden:

1. Directorio de trabajo actual. Es el directorio desde donde se invoca al compilador.
2. El directorio del sistema de Common Language Runtime.
3. Directorios especificados por **-lib**.
4. Directorios especificados por la variable de entorno LIB.

Use **-reference** para especificar una referencia a un ensamblado.

La opción **-lib** es sumatoria; si se especifica más de una vez, se anexa a valores ya existentes.

Una alternativa al uso de **-lib** consiste en copiar los ensamblados necesarios en el directorio de trabajo; esto permitirá pasar el nombre del ensamblado a **-reference**. Después, se pueden eliminar los ensamblados del directorio de trabajo. Dado que en el manifiesto del ensamblado no se especifica la ruta al ensamblado dependiente, la aplicación puede iniciarse en el equipo de destino y desde allí buscará y usará el ensamblado en la caché global de ensamblados.

El hecho de que el compilador puede hacer referencia al ensamblado no implica que Common Language Runtime pueda buscar y cargar el ensamblado en tiempo de ejecución. Vea Cómo el motor en tiempo de ejecución ubica ensamblados para obtener los detalles sobre cómo busca el motor en tiempo de ejecución los ensamblados a los que se hace referencia.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra el cuadro de diálogo **Páginas de propiedades** del proyecto.
2. Haga clic en la página de propiedades **Ruta de acceso de referencias**.

3. Modifique el contenido del cuadro de lista.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea ReferencePath.

Ejemplo

Compile t2.cs para crear un archivo .exe. El compilador buscará referencias a ensamblados en el directorio de trabajo y en el directorio raíz de la unidad C.

```
csc -lib:c:\ -reference:t2.dll t2.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-link (Opciones del compilador de C#)

29/06/2020 • 7 minutes to read • Edit Online

Hace que el compilador facilite al proyecto que se está compilando información de tipos COM en los ensamblados especificados.

Sintaxis

```
-link:fileList
// -or-
-l:fileList
```

Argumentos

`fileList`
Obligatorio. Lista delimitada por comas de nombres de archivos de ensamblado. Si el nombre de archivo contiene un espacio, escríbalo entre comillas.

Comentarios

La opción `-link` permite implementar una aplicación que tiene información de tipo incrustada. La aplicación puede usar los tipos de un ensamblado en tiempo de ejecución que implementan la información de tipo incrustada sin necesidad de una referencia al ensamblado en tiempo de ejecución. Si hay varias versiones del ensamblado en tiempo de ejecución publicadas, la aplicación que contiene la información de tipo incrustada puede trabajar con las distintas versiones sin tener que volver a compilar. Para obtener un ejemplo, vea Tutorial: Incrustar los tipos de los ensamblados administrados.

La opción `-link` resulta de especial utilidad cuando se trabaja con la interoperabilidad COM. Puede incrustar tipos COM para que la aplicación ya no necesite un ensamblado de interoperabilidad primario (PIA) en el equipo de destino. La opción `-link` indica al compilador que incruste la información de tipo COM del ensamblado de interoperabilidad al que se hace referencia en el código compilado resultante. El tipo COM se identifica mediante el valor de CLSID (GUID). Como resultado, la aplicación se puede ejecutar en un equipo de destino que tenga instalados los mismos tipos COM con los mismos valores de CLSID. Las aplicaciones que automatizan Microsoft Office son un buen ejemplo. Dado que las aplicaciones como Office suelen mantener el mismo valor de CLSID en las distintas versiones, la aplicación puede usar los tipos COM a los que se hace referencia siempre que .NET Framework 4 o posterior esté instalado en el equipo de destino y la aplicación emplee métodos, propiedades o eventos que estén incluidos en los tipos COM a los que se hace referencia.

La opción `-link` incrusta únicamente interfaces, estructuras y delegados. No se admite la incrustación de clases COM.

> **NOTE**
> Cuando se crea una instancia de un tipo COM incrustado en el código, hay que crear la instancia mediante la interfaz adecuada. Si se intenta crear una instancia de un tipo COM incrustado mediante la coclase, se produce un error.

Para establecer la opción `-link` en Visual Studio, agregue una referencia de ensamblado y establezca la propiedad `Embed Interop Types` en **true**. El valor predeterminado de la propiedad `Embed Interop Types` es **false**.

Si vincula a un ensamblado COM (ensamblado A) que a su vez hace referencia a otro ensamblado COM (ensamblado B), también debe vincular al ensamblado B si se cumple alguna de las siguientes condiciones:

- Un tipo del ensamblado A hereda de un tipo o implementa una interfaz del ensamblado B.
- Se invoca a un campo, una propiedad, un evento o un método que tiene un tipo de parámetro o un tipo de valor devuelto del ensamblado B.

Al igual que la opción del compilador -reference, la opción del compilador `-link` usa el archivo de respuesta Csc.rsp, que hace referencia a ensamblados de .NET Framework usados con frecuencia. Use la opción del compilador -noconfig si no quiere que el compilador emplee el archivo Csc.rsp.

La forma abreviada de `-link` es `-l`.

Tipos incrustados y genéricos

En las secciones siguientes se describen las limitaciones sobre el uso de tipos genéricos en aplicaciones que incrustan tipos de interoperabilidad.

Interfaces genéricas

No se puede usar interfaces genéricas que se incrusten desde un ensamblado de interoperabilidad. Esta implementación se muestra en el ejemplo siguiente.

```
// The following code causes an error if ISampleInterface is an embedded interop type.
ISampleInterface<SampleType> sample;
```

Tipos que tienen parámetros genéricos

Los tipos que tienen un parámetro genérico cuyo tipo se ha incrustado desde un ensamblado de interoperabilidad no se pueden usar si ese tipo pertenece a un ensamblado externo. Esta restricción no se aplica a las interfaces. Por ejemplo, considere la interfaz Range que se define en el ensamblado Microsoft.Office.Interop.Excel. Si una biblioteca inserta tipos de interoperabilidad desde el ensamblado Microsoft.Office.Interop.Excel y expone un método que devuelve un tipo genérico que tiene un parámetro cuyo tipo es la interfaz Range, ese método debe devolver una interfaz genérica, como se muestra en el ejemplo de código siguiente.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

En el ejemplo siguiente, el código de cliente puede llamar al método que devuelve la interfaz genérica IList sin errores.

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

Ejemplo

El código siguiente compila el archivo de origen `OfficeApp.cs` y hace referencia a ensamblados de `COMData1.dll` y `COMData2.dll` para generar `OfficeApp.exe`.

```
csc -link:COMData1.dll,COMData2.dll -out:OfficeApp.exe OfficeApp.cs
```

Vea también

- Opciones del compilador de C#
- Tutorial: Incrustación de los tipos de los ensamblados administrados
- -reference (Opciones del compilador de C#)
- -noconfig (Opciones del compilador de C#)
- Compilar la línea de comandos con csc.exe
- Información general sobre interoperabilidad

-linkresource (Opciones del compilador de C#)

18/03/2020 • 4 minutes to read • Edit Online

Crea un vínculo con un recurso de .NET Framework en el archivo de salida. El archivo de recursos no se agrega al archivo de salida. Esta opción difiere de la opción -resource, la cual sí inserta un archivo de recursos en el archivo de salida.

Sintaxis

```
-linkresource:filename[,identifier[,accessibility-modifier]]
```

Argumentos

`filename`
El archivo de recursos de .NET Framework con el que quiere crear un vínculo desde el ensamblado.

`identifier` (opcional)
El nombre lógico del recurso; nombre que se usa para cargar el recurso. El valor predeterminado es el nombre del archivo.

`accessibility-modifier` (opcional)
La accesibilidad del recurso: pública o privada. El valor predeterminado es public.

Comentarios

De manera predeterminada, los recursos vinculados son públicos en el ensamblado cuando se crean con el compilador de C#. Para que sean privados, especifique el modificador de accesibilidad `private`. No se permite ningún otro modificador distinto de `public` o `private`.

-linkresource requiere una de las opciones -target distinta de **-target:module**.

Si `filename` es un archivo de recursos de .NET Framework creado, por ejemplo, con Resgen.exe o en el entorno de desarrollo, se puede obtener acceso a él con miembros del espacio de nombres System.Resources. Para obtener más información, consulta System.Resources.ResourceManager. Para todos los demás recursos, use los métodos `GetManifestResource` de la clase Assembly para tener acceso al recurso en tiempo de ejecución.

El archivo especificado en `filename` puede tener cualquier formato. Por ejemplo, se puede hacer que una DLL nativa forme parte de un ensamblado para que se pueda instalar en la caché global de ensamblados y sea accesible desde código administrado del ensamblado. En el segundo de los siguientes ejemplos se muestra cómo hacerlo. También es posible realizar lo mismo en Assembly Linker. En el tercer ejemplo se muestra cómo hacerlo. Para obtener más información, vea Al.exe (Assembly Linker) y Trabajar con ensamblados y la memoria caché global de ensamblados.

-linkres es la forma abreviada de **-linkresource**.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Ejemplo

Para compilar `in.cs` y vincularlo al archivo de recursos `rf.resource`:

```
csc -linkresource:rf.resource in.cs
```

Ejemplo

Compile `A.cs` en un archivo DLL, vincúlelo a un archivo DLL nativo N.dll y coloque el resultado en la caché global de ensamblados (GAC). En este ejemplo, A.dll y N.dll residen en la GAC.

```
csc -linkresource:N.dll -t:library A.cs
gacutil -i A.dll
```

Ejemplo

Este ejemplo hace lo mismo que el anterior, pero usa las opciones de Assembly Linker.

```
csc -t:module A.cs
al -out:A.dll A.netmodule -link:N.dll
gacutil -i A.dll
```

Vea también

- Opciones del compilador de C#
- Al.exe (Assembly Linker)
- Trabajar con ensamblados y la memoria caché global de ensamblados
- Administrar propiedades de proyectos y de soluciones

-main (Opciones del compilador de C#)

29/06/2020 • 2 minutes to read • Edit Online

Esta opción especifica la clase que contiene el punto de entrada al programa si hay más de una clase que contenga un método **Main**.

Sintaxis

```
-main:class
```

Argumentos

`class`
Tipo que contiene el método **Main**.
El nombre de clase proporcionado debe ser completo: debe incluir el espacio de nombres completo que contiene la clase, seguido del nombre de clase. Por ejemplo, cuando el método `Main` se encuentra dentro de la clase `Program` en el espacio de nombres `MyApplication.Core`, la opción del compilador tiene que ser `-main:MyApplication.Core.Program`.

Comentarios

Si la compilación incluye más de un tipo con un método Main, puede especificar el tipo que contiene el método **Main** que quiere usar como punto de entrada en el programa.

Esta opción se usa al compilar un archivo .exe.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades **Aplicación**.
3. Modifique la propiedad **Objeto de inicio**.

   Para establecer esta opción del compilador mediante programación, vea StartupObject.

Para establecer esta opción del compilador editando manualmente el archivo. csproj

Puede establecer esta opción editando el archivo. csproj y agregando un elemento `StartupObject` dentro de la sección `PropertyGroup`. Por ejemplo:

```
<PropertyGroup>
  ...
  <StartupObject>MyApplication.Core.Program</StartupObject>
</PropertyGroup>
```

Ejemplo

Compile `t2.cs` y `t3.cs` y especifique que el método **Main** se encuentra en `Test2`:

```
csc t2.cs t3.cs -main:Test2
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de soluciones y proyectos

-moduleassemblyname (Opción del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Especifica un ensamblado con tipos no públicos a los que puede acceder un archivo .netmodule.

Sintaxis

```
-moduleassemblyname:assembly_name
```

Argumentos

`assembly_name`

El nombre del ensamblado a cuyos tipos no públicos .netmodule puede tener acceso.

Comentarios

Se debería usar **-moduleassemblyname** al compilar un .netmodule cuando se den las condiciones siguientes:

- .netmodule necesita tener acceso a los tipos no públicos de un ensamblado existente.
- Sabe el nombre del ensamblado en el que se compilará .netmodule.
- El ensamblado existente ha concedido acceso de ensamblado de confianza al ensamblado en el que se compilará .netmodule.

Para más información sobre cómo compilar .netmodule, vea -target:module (Opciones del compilador de C#).

Para obtener más información sobre los ensamblados de confianza, vea Ensamblados de confianza.

Esta opción no está disponible en el entorno de desarrollo; solo está disponible cuando se compila desde la línea de comandos.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Ejemplo

Este ejemplo compila un ensamblado con un tipo privado que concede acceso de ensamblado de confianza a un ensamblado llamado csman_an_assembly.

```
// moduleassemblyname_1.cs
// compile with: -target:library
using System;
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo ("csman_an_assembly")]

class An_Internal_Class
{
    public void Test()
    {
        Console.WriteLine("An_Internal_Class.Test called");
    }
}
```

Ejemplo

Este ejemplo compila un .netmodule que tiene acceso a un tipo no público del ensamblado moduleassemblyname_1.dll. Sabiendo que este .netmodule se integrará en un ensamblado denominado csman_an_assembly, se puede especificar **-moduleassemblyname**, lo que permite que .netmodule tenga acceso a los tipos no públicos de un ensamblado que ha concedido acceso de ensamblado de confianza a csman_an_assembly.

```
// moduleassemblyname_2.cs
// compile with: -moduleassemblyname:csman_an_assembly -target:module -reference:moduleassemblyname_1.dll
class B {
    public void Test() {
        An_Internal_Class x = new An_Internal_Class();
        x.Test();
    }
}
```

Ejemplo

Este ejemplo de código compila el ensamblado csman_an_assembly, haciendo referencia al ensamblado compilado previamente y a .netmodule.

```
// csman_an_assembly.cs
// compile with: -addmodule:moduleassemblyname_2.netmodule -reference:moduleassemblyname_1.dll
class A {
    public static void Main() {
        B bb = new B();
        bb.Test();
    }
}
```

Se ha llamado a An_Internal_Class.Test

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-noconfig (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-noconfig** comunica al compilador que no compile con el archivo csc.rsp, que está ubicado y se carga desde el mismo directorio que el archivo csc.exe.

Sintaxis

```
-noconfig
```

Comentarios

El archivo csc.rsp hace referencia a todos los ensamblados incluidos en .NET Framework. Las referencias reales que incluye el entorno de desarrollo Visual Studio de .NET dependen del tipo de proyecto.

Es posible modificar el archivo csc.rsp y especificar opciones de compiladores adicionales que puedan incluirse en cada compilación desde la línea de comandos con csc.exe (excepto la opción **-noconfig**).

El compilador procesa en último lugar las opciones que se pasan al comando **csc**. Por lo tanto, cualquier opción de la línea de comandos reemplaza la configuración de la misma opción en el archivo csc.rsp.

Si no quiere que el compilador busque y use la configuración del archivo csc.rsp, especifique **-noconfig**.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-nologo (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-nologo** suprime la presentación del banner de inicio de sesión cuando se inicia el compilador y muestra mensajes informativos durante la compilación.

Sintaxis

```
-nologo
```

Comentarios

Esta opción no está disponible en el entorno de desarrollo; solo está disponible cuando se compila desde la línea de comandos.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-nostdlib (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

-nostdlib impide la importación del archivo mscorlib.dll, que define el espacio de nombres System completo.

Sintaxis

```
-nostdlib[+ | -]
```

Comentarios

Use esta opción si desea definir o crear sus propios objetos y espacio de nombres System.

Si no se especifica **-nostdlib**, el archivo mscorlib.dll se importa en el programa (equivale a especificar **-nostdlib-**). Especificar **-nostdlib** es lo mismo que especificar **-nostdlib+** .

Para establecer esta opción del compilador en Visual Studio

> **NOTE**
>
> Las instrucciones siguientes solo se aplican a Visual Studio 2015 (y versiones anteriores). La propiedad de compilación **No hacer referencia a mscorlib.dll** no existe en Visual Studio.

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar** .
3. Haga clic en el botón **Avanzada** .
4. Modifique la propiedad **No hacer referencia al archivo mscorlib.dll** .

Para establecer esta opción del compilador mediante programación

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea NoStdLib.

Vea también

- Opciones del compilador de C#

-nowarn (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-nowarn** permite impedir que el compilador muestre una o más advertencias. Separe varios números de advertencia con una coma.

Sintaxis

```
-nowarn:number1[,number2,...]
```

Argumentos

`number1`, `number2`
Números de advertencia que quiere que el compilador suprima.

Comentarios

Solo debe especificar la parte numérica del identificador de advertencia. Por ejemplo, si quiere suprimir CS0028, puede especificar `-nowarn:28`.

El compilador omitirá silenciosamente los números de advertencia pasados a `-nowarn` que eran válidos en versiones anteriores, pero que se han quitado del compilador. Por ejemplo, CS0679 era válido en el compilador de Visual Studio .NET 2002, pero se ha eliminado posteriormente.

Las advertencias siguientes no se pueden suprimir mediante la opción `-nowarn`:

- Advertencia del compilador (nivel 1) CS2002
- Advertencia del compilador (nivel 1) CS2023
- Advertencia del compilador (nivel 1) CS2029

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto. Para obtener más información, vea Compilar (Página, Diseñador de proyectos) (C#).
2. Haga clic en la página de propiedades de **Compilar**.
3. Modifique la propiedad **Suprimir advertencias**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea DelaySign.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones
- Errores del compilador de C#

-nowin32manifest (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Use la opción **-nowin32manifest** para indicar al compilador que no inserte ningún manifiesto de aplicación en el archivo ejecutable.

Sintaxis

```
-nowin32manifest
```

Comentarios

Cuando se use esta opción, la aplicación estará sujeta a la virtualización en Windows Vista a menos que proporcione un manifiesto de aplicación en un archivo de recursos Win32 o durante un paso de compilación posterior.

En Visual Studio, defina esta opción en la página de **propiedades de la aplicación** seleccionando la opción **Crear aplicación sin manifiesto** en la lista desplegable **Manifiesto**. Para obtener más información, consulte Página de aplicación, Diseñador de proyectos (C#).

Para más información sobre la creación de manifiestos, vea -win32manifest (C# Compiler Options) (-win32manifest [Opciones del compilador de C#]).

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-nullable (opciones del compilador de C#)

29/06/2020 • 2 minutes to read • Edit Online

La opción **-nullable** le permite especificar el contexto que admite un valor NULL que quiera.

Sintaxis

```
-nullable[+ | -]
-nullable:{enable | disable | warnings | annotations}
```

Argumentos

`+` | `-`
Si se especifica `+`, o simplemente **-nullable**, el compilador habilitará el contexto que admite un valor NULL. Si se especifica `-`, que se aplica si no se especifica **-nullable**, se deshabilita el contexto que admite un valor NULL.

`enable` | `disable` | `warnings` | `annotations`
Especifica la opción de contexto que admite un valor NULL. Es similar a `+` o `-` en cuanto a habilitación y deshabilitación, pero permite una mayor granularidad de la especificidad del contexto que admite un valor NULL. El argumento `enable`, que se aplica del mismo modo que al especificar **-nullable**, habilita el contexto que admite un valor NULL. Si se especifica `disable`, se deshabilitará el contexto que admite un valor NULL. Al proporcionar el argumento `warnings`, **-nullable:warnings**, se habilita el contexto de advertencia que admite un valor NULL. Al especificar el argumento `annotations`, **-nullable:warnings**, se habilita el contexto de anotación que admite un valor NULL.

Comentarios

El análisis de flujo se utiliza para inferir la nulabilidad de las variables del código ejecutable. La nulabilidad inferida de una variable es independiente de la nulabilidad declarada de dicha variable. Las llamadas de métodos se analizan incluso cuando se omiten de forma condicional. Es el caso de Debug.Assert en modo de versión.

La invocación de métodos anotados con los siguientes atributos también afectará al análisis de flujo:

- Condiciones previas simples: AllowNullAttribute y DisallowNullAttribute
- Condiciones posteriores simples: MaybeNullAttribute y NotNullAttribute
- Condiciones posteriores condicionales: MaybeNullWhenAttribute y NotNullWhenAttribute
- DoesNotReturnIfAttribute (por ejemplo, `DoesNotReturnIf(false)` para Debug.Assert) y DoesNotReturnAttribute
- NotNullIfNotNullAttribute
- Condiciones posteriores de miembros: MemberNotNullAttribute(String) y MemberNotNullAttribute(String[])

Cómo establecer esta opción del compilador en un proyecto

Edite el archivo *.csproj* para agregar la etiqueta `<Nullable>` dentro de una jerarquía `Project/PropertyGroup`:

```
<Project Sdk="...">

  <PropertyGroup>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Vea también

- Opciones del compilador de C#
- Tipos de referencia que aceptan valores null

-optimize (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-optimize** habilita o deshabilita las optimizaciones realizadas por el compilador para que el archivo de salida sea menor, más rápido y más eficaz.

Sintaxis

```
-optimize[+ | -]
```

Comentarios

-optimize también indica a Common Language Runtime que optimice el código en tiempo de ejecución.

De forma predeterminada, las optimizaciones están deshabilitadas. Especifique **-optimize+** para habilitar las optimizaciones.

Al compilar un módulo para que lo use un ensamblado, use la misma configuración de **-optimize** que la del ensamblado.

-o es la forma abreviada de **-optimize**.

Es posible combinar las opciones **-optimize** y -debug.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar**.
3. Modifique la propiedad **Optimizar código**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea Optimize.

Ejemplo

Compile `t2.cs` y habilite las optimizaciones del compilador:

```
csc t2.cs -optimize
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-out (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

La opción **-out** especifica el nombre del archivo de salida.

Sintaxis

```
-out:filename
```

Argumentos

`filename`
El nombre del archivo de salida creado por el compilador.

Comentarios

En la línea de comandos, es posible especificar varios archivos de salida de una compilación. El compilador espera encontrar al menos un archivo de código fuente después de la opción **-out**. Posteriormente, todos los archivos de código fuente se compilarán en el archivo de salida especificado por la opción **-out**.

Hay que especificar el nombre completo y la extensión del archivo que se quiere crear.

Si no se especifica el nombre del archivo de salida:

- Un archivo .exe toma el nombre del archivo de código fuente que contiene el método **Main**.
- Un archivo .dll o .netmodule toma el nombre del primer archivo de código fuente.

Un archivo de código fuente usado para compilar un archivo de salida no puede usarse para compilar otro archivo de este tipo en la misma compilación.

Cuando se producen varios archivos de salida en una compilación de línea de comandos, recuerde que solo uno de los archivos de salida puede ser un ensamblado y que solo el primero que haya especificado (ya sea implícita o explícitamente con **-out**) puede ser el ensamblado.

Todos los módulos que se produzcan como parte de una compilación se convierten en archivos asociados a cualquier ensamblado que también se haya producido en la compilación. Use ildasm.exe para ver el manifiesto del ensamblado y los archivos asociados.

Es obligatorio usar la opción -out del compilador para que un archivo exe sea el destino de un ensamblado de confianza. Para obtener más información, vea Ensamblados de confianza.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades **Aplicación**.
3. Modifique la propiedad **Nombre del ensamblado**.

   Para establecer esta opción del compilador mediante programación: OutputFileName es una propiedad de solo lectura que se determina a partir de una combinación del tipo de proyecto (ejecutable, biblioteca, etc.) y el nombre del ensamblado. Es necesario modificar una de estas propiedades o ambas para establecer el nombre del archivo de salida.

Ejemplo

Compile `t.cs` y cree el archivo de salida `t.exe`, así como compile `t2.cs` y cree el archivo de salida del módulo `mymodule.netmodule`:

```
csc t.cs -out:mymodule.netmodule -target:module t2.cs
```

Vea también

- Opciones del compilador de C#
- Ensamblados de confianza
- Administrar propiedades de proyectos y de soluciones

-pathmap (opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-pathmap** del compilador especifica cómo asignar rutas físicas a los nombres de rutas de acceso de origen generados por el compilador.

Sintaxis

```
-pathmap:path1=sourcePath1,path2=sourcePath2
```

Argumentos

`path1` La ruta de acceso completa a los archivos de origen en el entorno actual

`sourcePath1` La ruta de acceso de origen sustituida por `path1` en cualquier archivo de salida

Para especificar varias rutas de acceso de origen asignadas, sepárelas con una coma.

Comentarios

El compilador escribe la ruta de acceso de origen en la salida por las razones siguientes:

1. La ruta de acceso de origen se sustituye por un argumento cuando se aplica CallerFilePathAttribute a un parámetro opcional.
2. La ruta de acceso de origen se inserta en un archivo PDB.
3. La ruta de acceso del archivo PDB se inserta en un archivo PE (ejecutable portable).

Esta opción asigna cada ruta de acceso física en la máquina donde el compilador se ejecuta a una ruta de acceso correspondiente que debe escribirse en los archivos de salida.

Ejemplo

Compile `t.cs` en el directorio **C:\work\test** y asigne ese directorio a **\publish** en la salida:

```
csc -pathmap:C:\work\tests=\publish t.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-pdb (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción del compilador **-pdb** especifica el nombre y la ubicación del archivo de símbolos de depuración.

Sintaxis

```
-pdb:filename
```

Argumentos

`filename`
El nombre y la ubicación del archivo de símbolos de depuración.

Comentarios

Al especificar -debug (Opciones del compilador de C#), el compilador creará un archivo .pdb en el mismo directorio en que el compilador creará el archivo de salida (.exe o .dll) con un nombre de archivo que es el mismo que el nombre del archivo de salida.

-pdb le permite especificar un nombre de archivo y una ubicación distintos del valor predeterminado para el archivo .pdb.

No se puede establecer esta opción del compilador en el entorno de desarrollo de Visual Studio, ni se puede cambiar mediante programación.

Ejemplo

Compile `t.cs` y cree un archivo .pdb denominado tt.pdb:

```
csc -debug -pdb:tt t.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-platform (Opciones del compilador de C#)

18/03/2020 • 4 minutes to read • Edit Online

Especifica qué versión de Common Language Runtime (CLR) puede ejecutar el ensamblado.

Sintaxis

```
-platform:string
```

Parámetros

`string`
anycpu (valor predeterminado), anycpu32bitpreferred, ARM, x64, x86 o Itanium.

Comentarios

- **anycpu** (valor predeterminado) compila el ensamblado de forma que se pueda ejecutar en cualquier plataforma. La aplicación se ejecuta como un proceso de 64 bits siempre que sea posible y recurre a 32 bits solo cuando el modo está disponible.
- **anycpu32bitpreferred** compila el ensamblado de forma que se pueda ejecutar en cualquier plataforma. La aplicación se ejecuta en modo de 32 bits en sistemas que admiten aplicaciones de 64 y 32 bits. Solo puede especificar esta opción para los proyectos que tienen como destino .NET Framework 4.5.
- **ARM** compila el ensamblado de forma que pueda ejecutarse en un equipo que tenga un procesador Advanced RISC Machine (ARM).
- **ARM64** compila el ensamblado que se va a ejecutar mediante el CLR de 64 bits en un equipo que tiene un procesador Advanced RISC Machine (ARM) que admite el conjunto de instrucciones A64.
- **x64** compila el ensamblado de forma que el CLR de 64 bits pueda ejecutarlo en equipos compatibles con el conjunto de instrucciones AMD64 o EM64T.
- **x86** compila el ensamblado de forma que el CLR de 32 bits compatible con x86 pueda ejecutarlo.
- **Itanium** compila el ensamblado de forma que el CLR de 64 bits pueda ejecutarlo en un equipo con un procesador Itanium.

En un sistema operativo de Windows de 64 bits:

- Los ensamblados compilados con **-platform:x86** se ejecutarán en el CLR de 32 bits que se ejecuta en WOW64.
- Un archivo DLL compilado con **-platform:anycpu** se ejecuta en el mismo CLR que el proceso en el que se ha cargado.
- Los archivos ejecutables que se compilan con **-platform:anycpu** se ejecutan en el CLR de 64 bits.
- Los archivos ejecutables compilados con **-platform:anycpu32bitpreferred** se ejecutan en el CLR de 32 bits.

La configuración **anycpu32bitpreferred** es válida solo para archivos ejecutables (.exe) y requiere .NET Framework 4.5.

Para obtener más información sobre cómo desarrollar una aplicación para que se ejecute en un sistema operativo Windows de 64 bits, consulte Aplicaciones de 64 bits.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar**.
3. Modifique la propiedad **Destino de la plataforma** y, para los proyectos que tienen como destino .NET Framework 4.5, active o desactive la casilla **Preferencia de 32 bits**.

> **NOTE**
>
> `-platform` no está disponible en el entorno de desarrollo de Visual C# Express.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea PlatformTarget.

Ejemplo

En el ejemplo siguiente, se muestra cómo usar la opción **-platform** para especificar que la aplicación se debe ejecutar en el CLR de 64 bits en un sistema operativo Windows de 64 bits.

```
csc -platform:anycpu filename.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-preferreduilang (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Mediante la opción del compilador `-preferreduilang`, puede especificar el idioma en el que el compilador de C# muestra el resultado, como los mensajes de error.

Sintaxis

```
-preferreduilang: language
```

Argumentos

`language`
El nombre del idioma del idioma que se va a usar para los resultados del compilador.

Comentarios

Puede usar la opción del compilador `-preferreduilang` para especificar el idioma que quiere que use el compilador de C# para los mensajes de error y otros resultados de la línea de comandos. Si el paquete de idioma para el idioma no está instalado, se usa la configuración del idioma del sistema operativo en su lugar, y no se notifica ningún error.

```
csc.exe -preferreduilang:ja-JP
```

Requisitos

Vea también

- Opciones del compilador de C#

-filealign (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Esta opción hace que el compilador aplique una clave pública pero no firma el ensamblado. La opción **-publicsign** también establece un bit en el ensamblado que indica el entorno de tiempo de ejecución en el que el archivo está firmado.

Sintaxis

```
-publicsign
```

Argumentos

Ninguno.

Comentarios

La opción **-publicsign** requiere el uso de -keyfile o -keycontainer. Las opciones **keyfile** o **keycontainer** especifican la clave pública.

Las opciones **-publicsign** y **-delaysign** son mutuamente excluyentes.

A veces llamada "firma falsa" o "firma OSS", la firma pública incluye la clave pública en un ensamblado de salida y establece la marca "firmado", pero en realidad no firma el ensamblado con una clave privada. Esto es útil para proyectos de código abierto donde la gente quiere crear ensamblados que sean compatibles con los ensamblados "totalmente firmados" publicados, pero que no tengan acceso a la clave privada utilizada para firmar los ensamblados. Dado que casi ningún consumidor necesita realmente comprobar si el ensamblado está completamente firmado, estos ensamblados públicamente creados se pueden utilizar en casi todos los escenarios en los que se utilizaría el ensamblado totalmente firmado.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Modifique la propiedad **Solo retrasar firma**.

Vea también

- Opción -delaysign del compilador de C#
- Opción -keyfile del compilador de C#
- Opción -keycontainer del compilador de C#
- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-recurse (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción -recurse permite compilar archivos de código fuente de todos los directorios secundarios del directorio especificado (dir) o del directorio del proyecto.

Sintaxis

```
-recurse:[dir\]file
```

Argumentos

`dir` (opcional)
El directorio en el que quiere que comience la búsqueda. Si no se especifica, la búsqueda comienza en el directorio del proyecto.

`file`
Los archivos que buscará. Se admiten los caracteres comodín.

Comentarios

La opción **-recurse** permite compilar archivos de código fuente de todos los directorios secundarios del directorio especificado (`dir`) o del directorio del proyecto.

Puede usar caracteres comodín en un nombre de archivo para compilar todos los archivos coincidentes en el directorio del proyecto sin usar **-recurse**.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Ejemplo

Compila todos los archivos de C# en el directorio actual:

```
csc *.cs
```

Compila todos los archivos de C# en el directorio dir1\dir2 y los directorios que haya por debajo y genera dir2.dll:

```
csc -target:library -out:dir2.dll -recurse:dir1\dir2\*.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-reference (Opciones del compilador de C#)

18/03/2020 • 6 minutes to read • Edit Online

La opción **-reference** hace que el compilador importe información de tipo public del archivo especificado al proyecto actual, lo que permite hacer referencia a metadatos de los archivos de ensamblado especificados.

Sintaxis

```
-reference:[alias=]filename
-reference:filename
```

Argumentos

`filename`
El nombre de un archivo que contiene un manifiesto del ensamblado. Para importar más de un archivo, incluya una opción **-reference** diferente para cada archivo.

`alias`
Un identificador de C# válido que representa un espacio de nombres raíz que contendrá todos los espacios de nombres del ensamblado.

Comentarios

Para importar desde más de un archivo, incluya una opción **-reference** para cada archivo.

Los archivos que se importen deben contener un manifiesto; el archivo de salida debe haberse compilado con una de las opciones -target distinta de -target:module.

-r es la forma abreviada de **-reference**.

Use -addmodule para importar metadatos de un archivo de salida que no contenga un manifiesto de ensamblado.

Si hace referencia a un ensamblado (ensamblado A) que hace referencia a otro ensamblado (ensamblado B), debe hacer referencia al ensamblado B si:

- Un tipo que se use del ensamblado A hereda de un tipo o implementa una interfaz del ensamblado B.
- Se invoca un campo, una propiedad, un evento o un método que tiene un tipo de parámetro o un tipo de valor devuelto del ensamblado B.

Use -lib para especificar el directorio en el que se encuentran una o varias de las referencias de ensamblado. En el tema **-lib** también se describen los directorios en los que el compilador busca ensamblados.

Para que el compilador reconozca un tipo de un ensamblado (y no de un módulo), debe obligársele a que resuelva el tipo, lo que se puede conseguir si se define una instancia del tipo. Existen otras formas de que el compilador resuelva nombres de tipos en un ensamblado; por ejemplo, si se hereda de un tipo de un ensamblado, el compilador reconocerá el nombre del tipo.

A veces es necesario hacer referencia a dos versiones diferentes del mismo componente desde un ensamblado. Para ello, use la subopción de alias en el modificador **-reference** de cada archivo para distinguir entre los dos archivos. Este alias se usará como calificador para el nombre del componente y se resolverá en el componente

de uno de los archivos.

De forma predeterminada se usa el archivo de respuesta (.rsp) csc, que hace referencia a los ensamblados de .NET Framework usados habitualmente. Use -noconfig si no quiere que el compilador use csc.rsp.

> **NOTE**
>
> En Visual Studio, use el cuadro de diálogo **Agregar referencia**. Para obtener más información, consulta How to: Add or Remove References By Using the Reference Manager. Para garantizar un comportamiento equivalente entre agregar referencias mediante `-reference` y agregar referencias mediante el cuadro de diálogo **Agregar referencia**, establezca la propiedad **Incrustar tipos de interoperabilidad** en **False** para el ensamblado que se va a agregar. El valor predeterminado de la propiedad es **True**.

Ejemplo

En este ejemplo se muestra cómo usar la característica alias externo.

Compile el archivo de origen e importe los metadatos desde `grid.dll` y `grid20.dll`, que se han compilado previamente. Los dos archivos DLL contienen versiones independientes del mismo componente y se usan dos **-reference** con opciones de alias para compilar el archivo de origen. Las opciones tiene este aspecto:

```
-reference:GridV1=grid.dll -reference:GridV2=grid20.dll
```

Esto configura los alias externos `GridV1` y `GridV2`, que se usan en el programa por medio de una instrucción `extern`:

```
extern alias GridV1;
extern alias GridV2;
// Using statements go here.
```

Una vez hecho esto, se puede hacer referencia al control de cuadrícula desde `grid.dll` si se antepone `GridV1` al nombre del control, de esta forma:

```
GridV1::Grid
```

Además, se puede hacer referencia al control de cuadrícula desde `grid20.dll` si se antepone `GridV2` al nombre del control, de esta forma:

```
GridV2::Grid
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-refout (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-refout** especifica una ruta de archivo donde el ensamblado de referencia debe mostrarse. Esto se traduce en `metadataPeStream` en la API de emisión. Esta opción corresponde a la propiedad de proyecto ProduceReferenceAssembly de MSBuild.

Sintaxis

```
-refout:filepath
```

Argumentos

`filepath` La ruta de archivo del ensamblado de referencia. Generalmente debe coincidir con el ensamblado principal. La convención recomendada (que se usa por MSBuild) es colocar el ensamblado de referencia en una subcarpeta "ref/" con relación al ensamblado principal.

Comentarios

Los ensamblados de referencia son un tipo especial de ensamblado que contiene solo la cantidad mínima de metadatos necesarios para representar la superficie de la API pública de la biblioteca. Incluyen declaraciones para todos los miembros que son significativos al hacer referencia a un ensamblado en las herramientas de compilación, pero excluyen todas las implementaciones de miembros y las declaraciones de miembros privados que no tienen ningún impacto observable en su contrato de API. Para obtener más información, consulte Ensamblados de referencia en la Guía de .NET.

Las opciones `-refout` y `-refonly` son mutuamente excluyentes.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-refonly (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-refonly** indica que un ensamblado de referencia debe mostrarse en lugar de un ensamblado de implementación, como el resultado principal. El parámetro `-refonly` deshabilita de forma automática la generación de archivos PDB, ya que los ensamblados de referencia no pueden ejecutarse. Esta opción corresponde a la propiedad de proyecto ProduceOnlyReferenceAssembly de MSBuild.

Sintaxis

```
-refonly
```

Comentarios

Los ensamblados de referencia son un tipo especial de ensamblado que contiene solo la cantidad mínima de metadatos necesarios para representar la superficie de la API pública de la biblioteca. Incluyen declaraciones para todos los miembros que son significativos al hacer referencia a un ensamblado en las herramientas de compilación, pero excluyen todas las implementaciones de miembros y las declaraciones de miembros privados que no tienen ningún impacto observable en su contrato de API. Para obtener más información, consulte Ensamblados de referencia en la Guía de .NET.

Las opciones `-refonly` y `-refout` son mutuamente excluyentes.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-resource (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Inserta el recurso especificado en el archivo de salida.

Sintaxis

```
-resource:filename[,identifier[,accessibility-modifier]]
```

Argumentos

`filename`
El archivo de recursos de .NET Framework que quiere insertar en el archivo de salida.

`identifier` (opcional)
El nombre lógico del recurso; nombre que se usa para cargar el recurso. El valor predeterminado es el nombre del archivo.

`accessibility-modifier` (opcional)
La accesibilidad del recurso: pública o privada. El valor predeterminado es public.

Comentarios

Use -linkresource para vincular un recurso a un ensamblado y no agregar el archivo de recursos al archivo de salida.

De manera predeterminada, los recursos son públicos en el ensamblado cuando se crean mediante el compilador de C#. Para que sean privados, especifique el modificador de accesibilidad `private`. No se permite ninguna otra accesibilidad distinta de `public` o `private`.

Si `filename` es un archivo de recursos de .NET Framework creado, por ejemplo, con Resgen.exe o en el entorno de desarrollo, se puede obtener acceso a él con miembros del espacio de nombres System.Resources. Para obtener más información, consulta System.Resources.ResourceManager. Para todos los demás recursos, use los métodos `GetManifestResource` de la clase Assembly para tener acceso al recurso en tiempo de ejecución.

-res es la forma abreviada de **-resource**.

El orden de los recursos en el archivo de salida se determina a partir del orden especificado en la línea de comandos.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Agregue un archivo de recursos al proyecto.
2. Seleccione el archivo que quiere insertar en el **Explorador de soluciones**.
3. Seleccione **Acción de compilación** para el archivo en la ventana **Propiedades**.
4. Establezca **Acción de compilación** en **Recurso incrustado**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea BuildAction.

Ejemplo

Compile `in.cs` y adjunte el archivo de recursos `rf.resource`:

```
csc -resource:rf.resource in.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-subsystemversion (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

Especifica la versión mínima del subsistema en la que se puede ejecutar el archivo ejecutable generado, lo que determina las versiones de Windows en las que se puede ejecutar el archivo ejecutable. Normalmente, esta opción garantiza que el archivo ejecutable pueda aprovechar las características de seguridad concretas que no están disponibles en versiones anteriores de Windows.

> **NOTE**
>
> Para especificar el subsistema en sí mismo, use la opción del compilador -target.

Sintaxis

```
-subsystemversion:major.minor
```

Parámetros

`major.minor`

La versión mínima requerida del subsistema, expresada en una notación de puntos para las versiones principales y secundarias. Por ejemplo, puede especificar que una aplicación no se puede ejecutar en un sistema operativo que sea anterior a Windows 7 si establece el valor de esta opción en 6.01, como se describe en la tabla que aparece más adelante en este tema. Debe especificar los valores de `major` y `minor` como números enteros.

Los ceros a la izquierda en la versión `minor` no cambian la versión, pero los ceros a la derecha sí. Por ejemplo, 6.1 y 6.01 hacen referencia a la misma versión, pero 6.10 hace referencia a una versión diferente. Se recomienda expresar la versión secundaria como dos dígitos para evitar confusiones.

Comentarios

En la tabla siguiente se enumeran las versiones de subsistema habituales de Windows.

VERSIÓN DE WINDOWS	VERSIÓN DE SUBSISTEMA
Windows 2000	5.00
Windows XP	5,01
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

Valores predeterminados

El valor predeterminado de la opción del compilador **-subsystemversion** depende de las condiciones de esta lista:

- El valor predeterminado es 6.02 si se establece cualquier opción del compilador en la siguiente lista:
  - /target:appcontainerexe
  - /target:winmdobj
  - -platform:arm
- El valor predeterminado es 6,00 si usa MSBuild, tiene como destino .NET Framework 4.5 y no ha configurado ninguna de las opciones del compilador que se han especificado anteriormente en esta lista.
- El valor predeterminado es 4.00 si no se cumple ninguna de las condiciones anteriores.

Establecer esta opción

Para establecer la opción del compilador **-subsystemversion** en Visual Studio, debe abrir el archivo .csproj y especificar un valor para la propiedad `SubsystemVersion` en el XML de MSBuild. No se puede establecer esta opción en el IDE de Visual Studio. Para obtener más información, consulte la sección "Valores predeterminados" que aparece más arriba en este tema o Propiedades comunes de proyectos de MSBuild.

Vea también

- Opciones del compilador de C#

-target (Opciones del compilador de C#)

27/04/2020 • 2 minutes to read • Edit Online

La opción del compilador **-target** se puede especificar en uno de estos cuatro formatos:

/target:appcontainerexe
Para crear un archivo .exe para aplicaciones de la Tienda Windows 8.x.

/target:exe
Para crear un archivo .exe.

/target:library
Para crear una biblioteca de código.

/target:module
Para crear un módulo.

/target:winexe
Para crear un programa de Windows.

/target:winmdobj
Para crear un archivo .winmdobj intermedio.

A menos que se especifique **-target:module**, **-target** hace que se coloque un manifiesto de ensamblado de .NET Framework en un archivo de salida. Para obtener más información, vea Ensamblados en .NET y Atributos comunes.

El manifiesto de ensamblado se coloca en el primer archivo de salida .exe de la compilación o en el primer archivo DLL si no hay ningún archivo de salida .exe. Por ejemplo, en la siguiente línea de comandos, el manifiesto se colocará en `1.exe`:

```
csc -out:1.exe t1.cs -out:2.netmodule t2.cs
```

El compilador crea solo un manifiesto de ensamblado por compilación. La información sobre todos los archivos de una compilación se coloca en el manifiesto de ensamblado. Todos los archivos de salida, excepto los creados con **-target:module**, pueden contener un manifiesto de ensamblado. Cuando se generan varios archivos de salida en la línea de comandos, solo se puede crear un manifiesto de ensamblado que debe ir en el primer archivo de salida especificado en la línea de comandos. Independientemente de cuál sea el primer archivo de salida (**-target:exe**, **-target:winexe**, **-target:library** o **-target:module**), los demás archivos de salida generados en la misma compilación deben ser módulos (**-target:module**).

Si crea un ensamblado, puede indicar que todo o parte del código es conforme a CLS mediante el atributo CLSCompliantAttribute.

```
// target_clscompliant.cs
[assembly:System.CLSCompliant(true)]   // specify assembly compliance

[System.CLSCompliant(false)]   // specify compliance for an element
public class TestClass
{
    public static void Main() {}
}
```

Para obtener más información sobre cómo establecer esta opción del compilador mediante programación, vea OutputType.

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones
- -subsystemversion (Opciones del compilador de C#)

-target:appcontainerexe (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Si usa la opción del compilador **-target:appcontainerexe**, este crea un archivo ejecutable de Windows (.exe) que se debe ejecutar en un contenedor de la aplicación. Esta opción equivale a -target:winexe, pero está diseñada para las aplicaciones de la Tienda Windows 8.x.

Sintaxis

```
-target:appcontainerexe
```

Comentarios

Para exigir que la aplicación se ejecute en un contenedor de la aplicación, esta opción establece un bit en el archivo portable ejecutable (PE). Cuando se establece ese bit, se produce un error si el método CreateProcess intenta iniciar el archivo ejecutable fuera de un contenedor de la aplicación.

A menos que use la opción -out, el archivo de salida toma el nombre del archivo de entrada que contiene el método Main.

Cuando se especifica esta opción en un símbolo del sistema, todos los archivos hasta la siguiente opción **-out** o **-target** se usan para crear el archivo ejecutable.

Para establecer esta opción del compilador en el IDE

1. En el **Explorador de soluciones**, abra el menú contextual del proyecto y después elija **Propiedades**.
2. En la pestaña **Aplicación**, en la lista **Tipo de resultado**, elija **Aplicación de la Tienda Windows**.

   Esta opción solo está disponible para las plantillas de aplicaciones de la Tienda Windows 8.x.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea OutputType.

Ejemplo

El comando siguiente compila `filename.cs` en un archivo ejecutable de Windows que solo se puede ejecutar en un contenedor de la aplicación.

```
csc -target:appcontainerexe filename.cs
```

Vea también

- -target (Opciones del compilador de C#)
- -target:winexe (Opciones del compilador de C#)
- Opciones del compilador de C#

-target:exe (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-target:winexe** hace que el compilador cree una aplicación de consola ejecutable (EXE).

Sintaxis

```
-target:exe
```

Comentarios

La opción **-target:exe** está en vigor de manera predeterminada. El archivo ejecutable se creará con la extensión .exe.

Use -target:winexe para crear un archivo ejecutable de un programa de Windows.

A menos que se especifique de otro modo con la opción -out, el nombre del archivo de salida toma el nombre del archivo de entrada que contiene el método Main.

Cuando se especifica en la línea de comandos, se usan todos los archivos hasta la siguiente opción **-out** o **-target:module** para crear el archivo .exe.

Solo se necesita un método **Main** en los archivos de código fuente que se compilan en un archivo .exe. La opción del compilador -main le permite especificar la clase que contiene el método **Main**, en aquellos casos en los que el código tenga más de una clase con un método **Main**.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades **Aplicación**.
3. Modifique la propiedad **Tipo de salida**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea OutputType.

Ejemplo

Cada una de las siguientes líneas de comandos compilará `in.cs` y creará `in.exe`:

```
csc -target:exe in.cs
csc in.cs
```

Vea también

- -target (Opciones del compilador de C#)
- Opciones del compilador de C#

-target:library (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-target:library** hace que el compilador cree una biblioteca de vínculos dinámicos (DLL) en lugar de un archivo ejecutable (EXE).

Sintaxis

```
-target:library
```

Comentarios

El archivo DLL se creará con la extensión .dll.

A menos que se especifique lo contrario con la opción -out, el archivo de salida adopta el nombre del primer archivo de entrada.

Cuando se especifica en la línea de comandos, se usan todos los archivos hasta la siguiente opción **-out** o **-target:module** para crear el archivo .dll.

Al compilar un archivo .dll, no es necesario un método Main.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades **Aplicación**.
3. Modifique la propiedad **Tipo de salida**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea OutputType.

Ejemplo

Compilación de `in.cs` y creación de `in.dll`:

```
csc -target:library in.cs
```

Vea también

- -target (Opciones del compilador de C#)
- Opciones del compilador de C#

-target:module (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

Esta opción hace que el compilador no genere un manifiesto del ensamblado.

Sintaxis

```
-target:module
```

Comentarios

De forma predeterminada, el archivo de salida creado al compilar con esta opción tendrá una extensión de .netmodule.

Common Language Runtime de .NET Framework no puede cargar un archivo que no tiene un manifiesto del ensamblado. En cambio, este archivo se puede incorporar en el manifiesto de un ensamblado mediante -addmodule.

Si se crea más de un módulo en una única compilación, los tipos internal que haya en un módulo estarán disponibles para otros módulos de la compilación. Cuando el código de un módulo hace referencia a tipos `internal` de otro módulo, se deben incorporar ambos módulos en un manifiesto del ensamblado, mediante **-addmodule**.

No se admite la creación de un módulo en el entorno de desarrollo de Visual Studio.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea OutputType.

Ejemplo

Compilación de `in.cs` y creación de `in.netmodule`:

```
csc -target:module in.cs
```

Vea también

- -target (Opciones del compilador de C#)
- Opciones del compilador de C#

-target:winexe (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-target:winexe** hace que el compilador cree un programa de Windows ejecutable (EXE).

Sintaxis

```
-target:winexe
```

Comentarios

El archivo ejecutable se creará con la extensión .exe. Un programa de Windows es aquel que ofrece una interfaz de usuario de la biblioteca de .NET Framework o con las API Windows.

Use -target:exe para crear una aplicación de consola.

A menos que se especifique de otro modo con la opción -out, el nombre del archivo de salida toma el nombre del archivo de entrada que contiene el método Main.

Cuando se especifica en la línea de comandos, se usan todos los archivos hasta la siguiente opción **-out** o -target para crear el programa de Windows.

Solo se necesita un método **Main** en los archivos de código fuente que se compilan en un archivo .exe. La opción -main permite especificar la clase que contiene el método **Main**, en aquellos casos en los que el código tenga más de una clase con un método **Main**.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades **Aplicación**.
3. Modifique la propiedad **Tipo de salida**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea OutputType.

Ejemplo

Compile `in.cs` en un programa de Windows:

```
csc -target:winexe in.cs
```

Vea también

- -target (Opciones del compilador de C#)
- Opciones del compilador de C#

-target:winmdobj (Opciones del compilador de C#)

18/03/2020 • 3 minutes to read • Edit Online

Si usa la opción del compilador **-target:winmdobj**, el compilador crea un archivo .winmdobj intermedio que se puede convertir en un archivo binario de Windows Runtime (.winmd). A continuación, pueden usar el archivo .winmd programas de JavaScript y C++, además de los programas de lenguajes administrados.

Sintaxis

```
-target:winmdobj
```

Comentarios

El valor **winmdobj** indica al compilador que un módulo intermedio es obligatorio. En respuesta, Visual Studio compila la biblioteca de clases de C# como archivo .winmdobj. El archivo .winmdobj se puede alimentar entonces a través de la herramienta de exportación WinMDExp para producir un archivo de metadatos de Windows (.winmd). El archivo .winmd contiene el código de la biblioteca original y los metadatos de WinMD que usan JavaScript o C++ y Windows en tiempo de ejecución.

La salida de un archivo compilado mediante la opción del compilador **-target:winmdobj** solo sirve para usarse como entrada de la herramienta de exportación WimMDExp; no se hace referencia directa al propio archivo .winmdobj.

A menos que use la opción -out, el nombre del archivo de salida adopta el nombre del primer archivo de entrada. No se requiere un método Main.

Si especifica la opción -target:winmdobj en un símbolo del sistema, todos los archivos hasta la siguiente opción **-out** o -target:module se usan para crear el programa de Windows.

Para establecer esta opción del compilador en el IDE de Visual Studio para una aplicación de la Tienda Windows

1. En el **Explorador de soluciones**, abra el menú contextual del proyecto y después elija **Propiedades**.
2. Pulse la pestaña **Aplicación**.
3. En la lista **Tipo de resultado**, pulse **Archivo WinMD**.

   La opción **Archivo WinMD** solo está disponible para las plantillas de aplicaciones de la Tienda Windows 8.x.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea OutputType.

Ejemplo

El comando siguiente compila `filename.cs` en un archivo .winmdobj intermedio.

```
csc -target:winmdobj filename.cs
```

Vea también

- -target (Opciones del compilador de C#)
- Opciones del compilador de C#

-unsafe (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción del compilador **-unsafe** permite la compilación de código en el que se usa la palabra clave unsafe.

Sintaxis

```
-unsafe
```

Comentarios

Para obtener más información sobre el código no seguro, vea Código no seguro y punteros.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar**.
3. Active la casilla **Permitir código no seguro**.

Para agregar esta opción en un archivo csproj

Abra el archivo .csproj de un proyecto y agregue los siguientes elementos:

```
<PropertyGroup>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea AllowUnsafeBlocks.

Ejemplo

Compile `in.cs` para el modo no seguro:

```
csc -unsafe in.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-utf8output (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-utf8output** muestra los resultados del compilador en codificación UTF-8.

Sintaxis

```
-utf8output
```

Comentarios

En algunas configuraciones internacionales, el resultado del compilador no puede mostrarse correctamente en la consola. En estas configuraciones, use **-utf8output** y redirija el resultado del compilador a un archivo.

Esta opción del compilador no está disponible en Visual Studio y no se puede cambiar mediante programación.

Vea también

- Opciones del compilador de C#

-warn (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-warn** especifica el nivel de advertencia que debe mostrar el compilador.

Sintaxis

```
-warn:option
```

Argumentos

`option`

El nivel de advertencia que quiere que se muestre para la compilación: los números más bajos muestran solo advertencias de gravedad alta; los números más altos muestran más advertencias. Los valores válidos son 0 a 4:

NIVEL DE ADVERTENCIA	SIGNIFICADO
0	Desactiva la emisión de todos los mensajes de advertencia.
1	Muestra mensajes de advertencia graves.
2	Muestra advertencias de nivel 1 además de determinadas advertencias menos graves, como advertencias sobre ocultar miembros de clase.
3	Muestra advertencias de nivel 2 además de determinadas advertencias menos graves, como advertencias sobre expresiones que siempre se evalúan como `true` o `false`.
4 (el valor predeterminado)	Muestra todas las advertencias de nivel 3 además de advertencias informativas.

Comentarios

Para obtener información sobre un error o advertencia, puede buscar el código de error en el Índice de la Ayuda. Para conocer otras maneras de obtener información sobre un error o advertencia, vea Errores del compilador de C#.

Use -warnaserror para tratar todas las advertencias como errores. Use -nowarn para deshabilitar determinadas advertencias.

-w es la forma abreviada de **-warn**.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar**.
3. Modifique la propiedad de **Nivel de advertencia**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea

WarningLevel.

Ejemplo

Compile `in.cs` y haga que el compilador solo muestre advertencias de nivel 1:

```
csc -warn:1 in.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-warnaserror (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-warnaserror+** trata todas las advertencias como errores

Sintaxis

```
-warnaserror[+ | -][:warning-list]
```

Comentarios

Cualquier mensaje que se notificaría de manera normal como una advertencia se notifica en su lugar como un error, y el proceso de compilación se detiene (no se compila ningún archivo de salida).

De manera predeterminada, **-warnaserror-** está en vigor, lo que provoca que las advertencias no impidan la generación de un archivo de salida. **-warnaserror**, que es igual que **-warnaserror+** , hace que las advertencias se traten como errores.

Opcionalmente, si solo quiere que algunas advertencias específicas se traten como errores, puede especificar una lista separada por comas de números de advertencia que se tratarán como errores. Se puede especificar el conjunto de todas las advertencias de nulabilidad con la abreviatura **nullable**.

Use -warn para especificar el nivel de advertencias que quiere que muestre el compilador. Use -nowarn para deshabilitar determinadas advertencias.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades de **Compilar**.
3. Modifique la propiedad **Tratar advertencias como errores**.

Para establecer esta opción del compilador mediante programación, vea TreatWarningsAsErrors.

Ejemplo

Compile `in.cs` y haga que el compilador no muestre ninguna advertencia:

```
csc -warnaserror in.cs
csc -warnaserror:642,649,652,nullable in.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-win32icon (Opciones del compilador de C#)

18/03/2020 • 2 minutes to read • Edit Online

La opción **-win32icon** inserta un archivo .ico en el archivo de salida, lo que proporciona al archivo de salida la apariencia esperada en el Explorador de archivos.

Sintaxis

```
-win32icon:filename
```

Argumentos

`filename`
El archivo .ico que quiere agregar a su archivo de salida.

Comentarios

Se puede crear un archivo .ico con el compilador de recursos. Cuando se compila un programa de Visual C++, se invoca el compilador de recursos y se crea un archivo .ico a partir del archivo .rc.

Vea -linkresource (para hacer referencia) o -resource (para adjuntar) un archivo de recursos de .NET Framework. Vea -win32res para importar un archivo .res.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra las páginas **Propiedades** del proyecto.
2. Haga clic en la página de propiedades **Aplicación**.
3. Modifique la propiedad **Icono de aplicación**.

Para obtener información sobre cómo establecer esta opción del compilador mediante programación, vea ApplicationIcon.

Ejemplo

Compile `in.cs` y adjunte un archivo .ico `rf.ico` para producir `in.exe`:

```
csc -win32icon:rf.ico in.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

-win32manifest (Opciones del compilador de C#)

18/03/2020 • 4 minutes to read • Edit Online

Use la opción **-win32manifest** para identificar un archivo de manifiesto de aplicación Win32 definido por el usuario que se va a incluir en un archivo portable ejecutable (PE) del proyecto.

Sintaxis

```
-win32manifest: filename
```

Argumentos

`filename`

Nombre y ubicación del archivo de manifiesto personalizado.

Comentarios

De forma predeterminada, el compilador de Visual C# inserta un manifiesto de aplicación que especifica un nivel de ejecución solicitado de "asInvoker". Crea el manifiesto en la misma carpeta en la que se ha compilado el archivo ejecutable, normalmente la carpeta bin\Debug o bin\Release cuando se usa Visual Studio. Si quiere proporcionar un manifiesto personalizado, por ejemplo para especificar un nivel de ejecución solicitado de "highestAvailable" o "requireAdministrator", use esta opción para especificar el nombre del archivo.

> **NOTE**
>
> Esta opción y -win32res (Opciones del compilador de C#) son mutuamente excluyentes. Si intenta usar ambas en la misma línea de comandos, obtendrá un error de compilación.

Una aplicación sin manifiesto de aplicación que especifique un nivel de ejecución solicitado estará sujeta a virtualización de archivos y Registro conforme a la característica Control de cuentas de usuario de Windows. Para más información, vea User Account Control (Control de cuentas de usuario).

La aplicación estará sujeta a virtualización si se cumple cualquiera de estas condiciones:

- Se usa la opción **-nowin32manifest** y no se proporciona ningún manifiesto en un paso de compilación posterior o como parte de un archivo de recursos de Windows (.res) mediante la opción **-win32res**.
- Se proporciona un manifiesto personalizado que no especifica un nivel de ejecución solicitado.

Visual Studio crea un archivo de manifiesto predeterminado y lo almacena en los directorios de depuración y versión junto con el archivo ejecutable. Puede agregar un manifiesto personalizado si crea uno en cualquier editor de texto y luego lo agrega al proyecto. También puede hacer clic con el botón derecho en el icono **Proyecto** del **Explorador de soluciones**, hacer clic en **Agregar nuevo elemento** y luego en **Archivo de manifiesto de aplicación**. Después de haber agregado el archivo de manifiesto nuevo o existente, aparecerá en la lista desplegable **Manifiesto**. Para obtener más información, consulte Página de aplicación, Diseñador de proyectos (C#).

Puede proporcionar el manifiesto de aplicación como un paso personalizado posterior a la compilación o como parte de un archivo de recursos Win32 mediante la opción -nowin32manifest (Opciones del compilador de C#). Use esa misma opción si quiere que la aplicación esté sujeta a virtualización de archivos y Registro en Windows

Vista. Esto evitará que el compilador cree e incruste un manifiesto predeterminado en el archivo portable ejecutable (PE).

Ejemplo

En el ejemplo siguiente se muestra el manifiesto predeterminado que el compilador de Visual C# inserta en un archivo PE.

> **NOTE**
> El compilador inserta un nombre de aplicación estándar "MyApplication.app" en el archivo xml. Se trata de una solución alternativa para permitir que las aplicaciones se ejecuten en Windows Server 2003 Service Pack 3.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

Vea también

- Opciones del compilador de C#
- -nowin32manifest (Opciones del compilador de C#)
- Administrar propiedades de proyectos y de soluciones

-win32res (Opciones del compilador de C#)

11/05/2020 • 2 minutes to read • Edit Online

La opción **-win32res** inserta un recurso de Win32 en el archivo de salida.

Sintaxis

```
-win32res:filename
```

Argumentos

`filename`
El archivo de recursos que quiere agregar a su archivo de salida.

Comentarios

Se puede crear un archivo de recursos de Win32 con el compilador de recursos. Cuando se compila un programa de Visual C++, se invoca el compilador de recursos y se crea un archivo .res a partir del archivo .rc.

Un recurso de Win32 puede contener información de versión o de mapa de bits (icono) que ayudaría a identificar la aplicación en el Explorador de archivos. Si no especifica **-win32res**, el compilador generará información de versión basada en la versión del ensamblado.

Vea -linkresource (para hacer referencia) o -resource (para adjuntar) un archivo de recursos de .NET Framework.

Para establecer esta opción del compilador en el entorno de desarrollo de Visual Studio

1. Abra la página **Propiedades** del proyecto.
2. Haga clic en la página de propiedades **Aplicación**.
3. Haga clic en el botón **Archivo de recursos** y seleccione un archivo mediante el cuadro combinado.

Ejemplo

Compile `in.cs` y adjunte un archivo de recursos de Win32 `rf.res` para producir `in.exe`:

```
csc -win32res:rf.res in.cs
```

Vea también

- Opciones del compilador de C#
- Administrar propiedades de proyectos y de soluciones

Errores del compilador de C#

18/03/2020 • 2 minutes to read • Edit Online

Algunos errores del compilador de C# tienen temas correspondientes que explican por qué se genera el error y, en algunos casos, cómo corregirlo. Utilice uno de los siguientes pasos para ver si la Ayuda está disponible para un mensaje de error concreto.

- Busque el número de error (por ejemplo, CS0029) en la Ventana de salida y después búsquelo en Microsoft Docs.
- Seleccione el número de error (por ejemplo, CS0029) en la Ventana de salida y después presione la tecla F1.
- En el índice, especifique el número de error en el cuadro **Buscar**.

Si ninguno de estos pasos conduce a información sobre el error, vaya al final de esta página y envíe comentarios con el número o el texto del error.

Para obtener información sobre cómo configurar las opciones de advertencia y error en C#, vea Compilar (Página, Diseñador de proyectos) (C#).

> **NOTE**
> Es posible que el equipo muestre nombres o ubicaciones diferentes para algunos de los elementos de la interfaz de usuario de Visual Studio en las siguientes instrucciones. La edición de Visual Studio que se tenga y la configuración que se utilice determinan estos elementos. Para obtener más información, vea Personalizar el IDE.

Vea también

- Opciones del compilador de C#
- No tenemos detalles sobre este error de C#
- Página Compilar (Diseñador de proyectos) (C#)
- -warn (Opciones del compilador de C#)
- -nowarn (Opciones del compilador de C#)

Introducción

29/06/2020 • 104 minutes to read • Edit Online

C# (pronunciado "si sharp" en inglés) es un lenguaje de programación sencillo, moderno, orientado a objetos y con seguridad de tipos. C#tiene sus raíces en la familia de lenguajes C y será inmediatamente familiar para los programadores de C, C++y Java. C#está normalizado por ECMA International como el estándar ***ECMA-334*** y por ISO/IEC como el estándar ***iso/IEC 23270*** . El C# compilador de Microsoft para el .NET Framework es una implementación compatible de ambos estándares.

C# es un lenguaje orientado a objetos, pero también incluye compatibilidad para programación ***orientada a componentes***. El diseño de software contemporáneo se basa cada vez más en componentes de software en forma de paquetes independientes y autodescriptivos de funcionalidad. La clave de estos componentes es que presentan un modelo de programación con propiedades, métodos y eventos; tienen atributos que proporcionan información declarativa sobre el componente; e incorporan su propia documentación. C#proporciona construcciones de lenguaje para admitir directamente estos conceptos, con C# lo que se crea un lenguaje muy natural en el que crear y usar componentes de software.

Varias características de C# ayudan en la construcción de aplicaciones sólidas y duraderas: La ***recolección de elementos*** no utilizados reclama automáticamente la memoria ocupada por objetos no usados. el ***control de excepciones*** proporciona un enfoque estructurado y extensible para la detección y recuperación de errores. y el diseño ***con seguridad de tipos*** del lenguaje hace que sea imposible leer desde variables no inicializadas, indizar matrices más allá de sus límites o realizar conversiones de tipo no comprobadas.

C# tiene un ***sistema de tipo unificado***. Todos los tipos de C#, incluidos los tipos primitivos como `int` y `double`, se heredan de un único tipo `object` raíz. Por lo tanto, todos los tipos comparten un conjunto de operaciones comunes, y los valores de todos los tipos se pueden almacenar, transportar y utilizar de manera coherente. Además, C# admite tipos de valor y tipos de referencia definidos por el usuario, lo que permite la asignación dinámica de objetos, así como almacenamiento en línea de estructuras ligeras.

Para asegurarse de C# que los programas y las bibliotecas pueden evolucionar con el tiempo de una manera compatible, se ha hecho mucho C#hincapié en el control de ***versiones*** en el diseño de. Muchos lenguajes de programación prestan poca atención a este problema y, como resultado, los programas escritos en dichos lenguajes se interrumpen con más frecuencia de lo necesario cuando se introducen nuevas versiones de las bibliotecas dependientes. Los aspectos C#del diseño de que se vieron afectados directamente por las consideraciones sobre el `virtual` control `override` de versiones son los modificadores y independientes, las reglas para la resolución de sobrecarga de métodos y la compatibilidad con las

declaraciones explícitas de miembros de interfaz.

En el resto de este capítulo se describen las características esenciales C# del lenguaje. Aunque los capítulos posteriores describen las reglas y las excepciones en un modo orientado a los detalles y, a veces, de forma matemática, en este capítulo se realiza una mayor claridad y una brevedad a costa de la integridad. La intención es proporcionar al lector una introducción al lenguaje que facilitará la escritura de programas iniciales y la lectura de capítulos posteriores.

Hola a todos

El programa "Hola mundo" tradicionalmente se usa para presentar un lenguaje de programación. En este caso, se usa C#:

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

Normalmente, los archivos de código fuente de C# tienen la extensión de archivo `.cs`. Suponiendo que el programa "Hola mundo" se almacena en el archivo `hello.cs`, el programa se puede compilar con el compilador de Microsoft C# mediante la línea de comandos.

```
csc hello.cs
```

que genera un ensamblado ejecutable denominado `hello.exe`. La salida generada por esta aplicación cuando se ejecuta es

```
Hello, World
```

El programa "Hola mundo" empieza con una directiva `using` que hace referencia al espacio de nombres `System`. Los espacios de nombres proporcionan un método jerárquico para organizar las bibliotecas y los programas de C#. Los espacios de nombres contienen tipos y otros espacios de nombres; por ejemplo, el espacio de nombres `System` contiene varios tipos, como la clase `Console` a la que se hace referencia en el programa, y otros espacios de nombres, como `IO` y `Collections`. Una directiva `using` que hace referencia a un espacio de nombres determinado permite el uso no calificado de los tipos que son miembros de ese espacio de nombres. Debido a la directiva `using`, puede utilizar el programa `Console.WriteLine` como abreviatura de `System.Console.WriteLine`.

La clase `Hello` declarada por el programa "Hola mundo" tiene un miembro único, el método llamado `Main`. El `Main` método se declara con el `static` modificador. Mientras que los métodos de instancia pueden hacer referencia a una instancia de objeto envolvente determinada utilizando la palabra clave `this`, los métodos estáticos funcionan sin referencia a un

objeto determinado. Por convención, un método estático denominado `Main` sirve como punto de entrada de un programa.

La salida del programa la genera el método `WriteLine` de la clase `Console` en el espacio de nombres `System`. Esta clase la proporcionan las bibliotecas de clases de .NET Framework, a las que, de forma predeterminada, el compilador de Microsoft C# hace referencia automáticamente. Tenga en C# cuenta que no tiene una biblioteca en tiempo de ejecución independiente. En su lugar, el .NET Framework es la biblioteca en C#tiempo de ejecución de.

Estructura del programa

Los principales conceptos organizativos en C# son ***programas***, ***espacios de nombres***, ***tipos***, ***miembros*** y ***ensamblados***. Los programas de C# constan de uno o más archivos de origen. Los programas declaran tipos, que contienen miembros y pueden organizarse en espacios de nombres. Las clases e interfaces son ejemplos de tipos. Los campos, los métodos, las propiedades y los eventos son ejemplos de miembros. Cuando se compilan programas de C#, se empaquetan físicamente en ensamblados. Normalmente, los ensamblados tienen `.exe` la `.dll` extensión de archivo o, dependiendo de si implementan ***aplicaciones*** o ***bibliotecas***.

El ejemplo

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

declara una clase denominada `Stack` en un espacio de nombres denominado. `Acme.Collections` El nombre completo de esta clase es `Acme.Collections.Stack`. La clase contiene varios miembros: un campo

denominado `top`, dos métodos denominados `Push` y `Pop`, y una clase anidada denominada `Entry`. La clase `Entry` contiene además tres miembros: un campo denominado `next`, un campo denominado `data` y un constructor. Suponiendo que el código fuente del ejemplo se almacene en el archivo `acme.cs`, la línea de comandos

```
csc /t:library acme.cs
```

compila el ejemplo como una biblioteca (código sin un punto de entrada `Main` y genera un ensamblado denominado `acme.dll`.

Los ensamblados contienen código ejecutable en forma de instrucciones de ***lenguaje intermedio*** (IL) e información simbólica en forma de ***metadatos***. Antes de ejecutarlo, el código de IL de un ensamblado se convierte automáticamente en código específico del procesador mediante el compilador de Just in Time (JIT) de .NET Common Language Runtime.

Dado que un ensamblado es una unidad autodescriptiva de funcionalidad que contiene código y metadatos, no hay necesidad de directivas `#include` ni archivos de encabezado de C#. Los tipos y miembros públicos contenidos en un ensamblado determinado estarán disponibles en un programa de C# simplemente haciendo referencia a dicho ensamblado al compilar el programa. Por ejemplo, este programa usa la clase `Acme.Collections.Stack` desde el ensamblado `acme.dll`:

```
using System;
using Acme.Collections;

class Test
{
    static void Main() {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

Si el programa se almacena en el archivo `test.cs`, cuando `test.cs` se compila, `acme.dll` se puede hacer referencia al ensamblado mediante la opción `/r` del compilador:

```
csc /r:acme.dll test.cs
```

Esto crea un ensamblado ejecutable denominado `test.exe`, que, cuando se ejecuta, produce el resultado:

```
100
10
1
```

C# permite que el texto de origen de un programa se almacene en varios

archivos de origen. Cuando se compila un programa de C# de varios archivos, todos los archivos de origen se procesan juntos y los archivos de origen pueden hacerse referencia mutuamente de forma libre; desde un punto de vista conceptual, es como si todos los archivos de origen estuvieran concatenados en un archivo de gran tamaño antes de ser procesados. En C# nunca se necesitan declaraciones adelantadas porque, excepto en contadas ocasiones, el orden de declaración es insignificante. C# no limita un archivo de origen a declarar solamente un tipo público ni precisa que el nombre del archivo de origen coincida con un tipo declarado en el archivo de origen.

Tipos y variables

Hay dos clases de tipos en C#: ***tipos de valor*** y ***tipos de referencia***. Las variables de tipos de valor contienen directamente los datos, mientras que las variables de los tipos de referencia almacenan referencias a los datos, lo que se conoce como objetos. Con los tipos de referencia, es posible que dos variables hagan referencia al mismo objeto y que, por tanto, las operaciones en una variable afecten al objeto al que hace referencia la otra variable. Con los tipos de valor, cada variable tiene su propia copia de los datos y no es posible que las operaciones en una variable afecten a la otra (excepto en el caso de las variables de parámetro `ref` y `out`).

C#los tipos de valor de se dividen en ***tipos simples***, ***tipos de enumeración***, ***tipos de struct***y tipos que C# ***aceptan valores NULL***, y los tipos de referencia de se dividen en ***tipos de clase***, ***tipos de interfaz***, ***matriz tipos***y ***tipos de delegado***.

En la tabla siguiente se proporciona información C#general sobre el sistema de tipos de.

CATEGORÍA		DESCRIPCIÓN
Tipos de valor	Tipos simples	Entero con signo: `sbyte`, `short`, `int`, `long`
		Entero sin signo: `byte`, `ushort`, `uint`, `ulong`
		Caracteres Unicode: `char`
		Punto flotante de IEEE: `float`, `double`
		Decimal de alta precisión: `decimal`
		Booleano: `bool`
	Tipos de enumeración	Tipos definidos por el usuario con el formato `enum E {...}`

CATEGORÍA		DESCRIPCIÓN
	Tipos de estructura	Tipos definidos por el usuario con el formato `struct S {...}`
	Tipos que aceptan valores NULL	Extensiones de todos los demás tipos de valor con un valor `null`
Tipos de referencia	Tipos de clase	Clase base definitiva de todos los demás tipos: `object`
		Cadenas Unicode: `string`
		Tipos definidos por el usuario con el formato `class C {...}`
	Tipos de interfaz	Tipos definidos por el usuario con el formato `interface I {...}`
	Tipos de matriz	Unidimensional y multidimensional; por ejemplo, `int[]` y `int[,]`
	Tipos delegados	Tipos definidos por el usuario del formulario, por ejemplo, `delegate int D(...)`

Los ocho tipos enteros proporcionan compatibilidad con valores de 8, 16, 32 y 64 bits en formato con o sin signo.

Los dos tipos de punto flotante `float` , `double` y, se representan mediante los formatos IEEE 754 de precisión sencilla de 32 bits y de doble precisión de 64 bits.

El tipo `decimal` es un tipo de datos de 128 bits adecuado para cálculos financieros y monetarios.

C#el `bool` tipo de se usa para representar valores booleanos: valores que `true` son `false` o.

El procesamiento de caracteres y cadenas en C# utiliza la codificación Unicode. El tipo `char` representa una unidad de código UTF-16 y el tipo `string` representa una secuencia de unidades de código UTF-16.

En la tabla siguiente se C#resumen los tipos numéricos de.

CATEGORÍA	PARADA	TIPO	INTERVALO/PRECISIÓN
Entero con signo	8	`sbyte`	-128... 127
	16	`short`	-32768... 32, 767
	32	`int`	-2147483648... 2, 147, 483, 647
	64	`long`	-9223372036854775808... 9, 223, 372, 036, 854, 775, 807
Entero sin signo	8	`byte`	0... 255
	16	`ushort`	0... 65, 535
	32	`uint`	0... 4, 294, 967, 295
	64	`ulong`	0... 18, 446, 744, 073, 709, 551, 615
Punto flotante	32	`float`	1,5 × 10 ^ − 45 a 3,4 × 10 ^ 38, precisión de 7 dígitos
	64	`double`	5,0 × 10 ^ − 324 a 1,7 × 10 ^ 308, precisión de 15 dígitos
Decimal	128	`decimal`	1,0 × 10 ^ − 28 a 7,9 × 10 ^ 28, precisión de 28 dígitos

Los programas de C# utilizan ***declaraciones de tipos*** para crear nuevos tipos. Una declaración de tipos especifica el nombre y los miembros del nuevo tipo. Cinco de C#las categorías de tipos de son definibles por el usuario: tipos de clase, tipos de struct, tipos de interfaz, tipos de enumeración y tipos de delegado.

Un tipo de clase define una estructura de datos que contiene miembros de datos (campos) y miembros de función (métodos, propiedades, etc.). Los tipos de clase admiten herencia única y polimorfismo, mecanismos por los que las clases derivadas pueden extender y especializar clases base.

Un tipo de estructura es similar a un tipo de clase en que representa una estructura con miembros de datos y miembros de función. Sin embargo, a diferencia de las clases, las estructuras son tipos de valor y no requieren la

asignación del montón. Los tipos struct no admiten la herencia especificada por el usuario y todos los tipos de struct se heredan implícitamente del tipo `object`.

Un tipo de interfaz define un contrato como un conjunto con nombre de miembros de función públicos. Una clase o estructura que implementa una interfaz debe proporcionar implementaciones de los miembros de función de la interfaz. Una interfaz puede heredar de varias interfaces base, y una clase o estructura puede implementar varias interfaces.

Un tipo de delegado representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son similares al concepto de punteros de función en otros lenguajes, pero a diferencia de los punteros de función, los delegados están orientados a objetos y presentan seguridad de tipos.

Los tipos de clase, estructura, interfaz y delegado admiten genéricos, mediante los cuales se pueden parametrizar con otros tipos.

Un tipo de enumeración es un tipo distinto con constantes con nombre. Cada tipo de enumeración tiene un tipo subyacente, que debe ser uno de los ocho tipos enteros. El conjunto de valores de un tipo de enumeración es igual que el conjunto de valores del tipo subyacente.

C# admite matrices unidimensionales y multidimensionales de cualquier tipo. A diferencia de los tipos enumerados anteriormente, los tipos de matriz no tienen que ser declarados antes de usarlos. En su lugar, los tipos de matriz se crean mediante un nombre de tipo entre corchetes. Por ejemplo, `int[]` es una matriz unidimensional de `int`, `int[,]` es una matriz bidimensional de `int` y `int[][]` es una matriz unidimensional de Matrices unidimensionales de `int`.

Los tipos que aceptan valores NULL tampoco tienen que declararse antes de que se puedan utilizar. Para cada tipo `T` de valor que no acepta valores NULL, existe un tipo `T?` que acepta valores NULL correspondiente, que puede `null` contener un valor adicional. Por ejemplo, `int?` es un tipo que puede contener cualquier entero de 32 bits o el valor `null`.

C#el sistema de tipos de está unificado, de modo que un valor de cualquier tipo se puede tratar como un objeto. Todos los tipos de C# directa o indirectamente se derivan del tipo de clase `object`, y `object` es la clase base definitiva de todos los tipos. Los valores de tipos de referencia se tratan como objetos mediante la visualización de los valores como tipo `object`. Los valores de los tipos de valor se tratan como objetos realizando operaciones de ***conversión boxing*** y ***unboxing*** . En el ejemplo siguiente, un valor `int` se convierte en `object` y vuelve a `int`.

```
using System;

class Test
{
    static void Main() {
        int i = 123;
        object o = i;          // Boxing
        int j = (int)o;        // Unboxing
    }
}
```

Cuando un valor de un tipo de valor se convierte al `object` tipo, se asigna una instancia de objeto, también denominada "box", para contener el valor, y el valor se copia en ese cuadro. Por el contrario, cuando una `object` referencia se convierte en un tipo de valor, se realiza una comprobación de que el objeto al que se hace referencia es un cuadro del tipo de valor correcto y, si la comprobación se realiza correctamente, se copia el valor en el cuadro.

C#el sistema de tipos unificado significa de forma eficaz que los tipos de valor pueden convertirse en objetos "a petición". Debido a la unificación, las bibliotecas de uso general que utilizan el tipo `object` pueden usarse con tipos de referencia y tipos de valor.

Hay varios tipos de ***variables*** en C#, entre otras, campos, elementos de matriz, variables locales y parámetros. Las variables representan ubicaciones de almacenamiento, y cada variable tiene un tipo que determina qué valores se pueden almacenar en la variable, como se muestra en la tabla siguiente.

TIPO DE VARIABLE	CONTENIDO POSIBLE
Tipo de valor distinto a NULL	Un valor de ese tipo exacto
Tipos de valor NULL	Un valor null o un valor de ese tipo exacto
`object`	Una referencia nula, una referencia a un objeto de cualquier tipo de referencia o una referencia a un valor de conversión boxing de cualquier tipo de valor
Tipo de clase	Una referencia nula, una referencia a una instancia de ese tipo de clase o una referencia a una instancia de una clase derivada de ese tipo de clase.
Tipo de interfaz	Una referencia nula, una referencia a una instancia de un tipo de clase que implementa ese tipo de interfaz o una referencia a un valor de conversión boxing de un tipo de valor que implementa ese tipo de interfaz.

TIPO DE VARIABLE	CONTENIDO POSIBLE
Tipo de matriz	Una referencia nula, una referencia a una instancia de ese tipo de matriz o una referencia a una instancia de un tipo de matriz compatible
Tipo delegado	Una referencia nula o una referencia a una instancia de ese tipo de delegado.

Expresiones

Las ***expresiones*** se construyen con ***operandos*** y ***operadores***. Los operadores de una expresión indican qué operaciones se aplican a los operandos. Ejemplos de operadores incluyen `+`, `-`, `*`, `/` y `new`. Algunos ejemplos de operandos son literales, campos, variables locales y expresiones.

Cuando una expresión contiene varios operadores, la ***precedencia*** de los operadores controla el orden en que se evalúan los operadores individuales. Por ejemplo, la expresión `x + y * z` se evalúa como `x + (y * z)` porque el operador `*` tiene mayor precedencia que el operador `+`.

La mayoría de los operadores se pueden ***sobrecargar***. La sobrecarga de operador permite la especificación de implementaciones de operadores definidas por el usuario para operaciones donde uno o ambos operandos son de un tipo de struct o una clase definidos por el usuario.

En la tabla siguiente se C#resumen los operadores de, que muestran las categorías de operador en orden de prioridad, de mayor a menor. Los operadores de la misma categoría tienen la misma precedencia.

CATEGORÍA	EXPRESIÓN	DESCRIPCIÓN
Principal	`x.m`	Acceso a miembros
	`x(...)`	Invocación de método y delegado
	`x[...]`	Acceso a matriz e indizador
	`x++`	Postincremento
	`x--`	Postdecremento
	`new T(...)`	Creación de objetos y delegados
	`new T(...){...}`	creación de objetos con inicializador

CATEGORÍA	EXPRESIÓN	DESCRIPCIÓN
	`new {...}`	inicializador de objetos anónimos
	`new T[...]`	creación de matriz
	`typeof(T)`	obtener el objeto `System.Type` para `T`
	`checked(x)`	Evaluar expresión en contexto comprobado
	`unchecked(x)`	Evaluar expresión en contexto no comprobado
	`default(T)`	obtener valor predeterminado de tipo `T`
	`delegate {...}`	Función anónima (método anónimo)
Unario	`+x`	identidad
	`-x`	Negación
	`!x`	Negación lógica
	`~x`	Negación bit a bit
	`++x`	Preincremento
	`--x`	Predecremento
	`(T)x`	convertir explícitamente `x` en el tipo `T`
	`await x`	esperar asincrónicamente a que finalice `x`
Multiplicativo	`x * y`	Multiplicación
	`x / y`	División
	`x % y`	Resto
Aditivo	`x + y`	Suma, concatenación de cadenas, combinación de delegados
	`x - y`	Resta, eliminación de delegados

CATEGORÍA	EXPRESIÓN	DESCRIPCIÓN
Shift	`x << y`	Desplazamiento a la izquierda
	`x >> y`	Desplazamiento a la derecha
Comprobación de tipos y relacional	`x < y`	Menor que
	`x > y`	Mayor que
	`x <= y`	Menor o igual que
	`x >= y`	Mayor o igual que
	`x is T`	volver a ejecutar `true` si `x` es una `T`, de lo contrario `false`
	`x as T`	volver a ejecutar `x` con tipo `T`, o `null` si `x` no es una `T`
Igualdad	`x == y`	Igual
	`x != y`	No igual
AND lógico	`x & y`	AND bit a bit entero, AND lógico booleano
XOR lógico	`x ^ y`	XOR bit a bit entero, XOR lógico booleano
OR lógico	`x | y`	OR bit a bit entero, OR lógico booleano
AND condicional	`x && y`	Solo evalúa si `x` es `y``true`
OR condicional	`x || y`	Solo evalúa si `x` es `y``false`
Uso combinado de NULL	`x ?? y`	Se evalúa como `y` si `x` es `null`, en `x` caso contrario,.
Condicional	`x ? y : z`	se evalúa como `y` si `x` es `true` o `z` si `x` es `false`
Asignación o función anónima	`x = y`	Asignación

CATEGORÍA	EXPRESIÓN	DESCRIPCIÓN
	`x op= y`	Asignación compuesta; los operadores admitidos son `*=` `/=` `%=` `+=` `-=` `<<=` `>>=` `&=` `^=` `|=`
	`(T x) => y`	Función anónima (expresión lambda)

Instrucciones

Las acciones de un programa se expresan mediante ***instrucciones***. C# admite varios tipos de instrucciones diferentes, varias de las cuales se definen en términos de instrucciones insertadas.

Un ***bloque*** permite que se escriban varias instrucciones en contextos donde se permite una única instrucción. Un bloque se compone de una lista de instrucciones escritas entre los delimitadores `{` y `}`.

Las ***instrucciones de declaración*** se usan para declarar variables locales y constantes.

Las ***instrucciones de expresión*** se usan para evaluar expresiones. Entre las expresiones que se pueden utilizar como instrucciones se incluyen las invocaciones de método, las asignaciones `new` de objetos que usan el operador `=` , las asignaciones mediante y los operadores de asignación compuesta, las operaciones de incremento y decremento mediante el `++` operadores `--` y y expresiones Await.

Las ***instrucciones de selección*** se usan para seleccionar una de varias instrucciones posibles para su ejecución en función del valor de alguna expresión. En este grupo están las instrucciones `if` y `switch`.

Las ***instrucciones de iteración*** se utilizan para ejecutar repetidamente una instrucción incrustada. En este grupo están las instrucciones `while`, `do`, `for` y `foreach`.

Las ***instrucciones de salto*** se usan para transferir el control. En este grupo están las instrucciones `break`, `continue`, `goto`, `throw`, `return` y `yield`.

La instrucción `try`... `catch` se usa para detectar excepciones que se producen durante la ejecución de un bloque, y la instrucción `try`... `finally` se usa para especificar el código de finalización que siempre se ejecuta, tanto si se ha producido una excepción como si no.

Las `checked` instrucciones `unchecked` y se utilizan para controlar el contexto de comprobación de desbordamiento para conversiones y operaciones aritméticas de tipo entero.

La instrucción `lock` se usa para obtener el bloqueo de exclusión mutua para un objeto determinado, ejecutar una instrucción y, luego, liberar el bloqueo.

La instrucción `using` se usa para obtener un recurso, ejecutar una instrucción y, luego, eliminar dicho recurso.

A continuación se muestran ejemplos de cada tipo de instrucción

Declaraciones de variables locales

```
static void Main() {
   int a;
   int b = 2, c = 3;
   a = 1;
   Console.WriteLine(a + b + c);
}
```

Declaración de constante local

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

Expression (instrucción)

```
static void Main() {
    int i;
    i = 123;                // Expression statement
    Console.WriteLine(i);   // Expression statement
    i++;                    // Expression statement
    Console.WriteLine(i);   // Expression statement
}
```

Instrucción `if`

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

Instrucción `switch`

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

Instrucción `while`

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

Instrucción `do`

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

Instrucción `for`

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

Instrucción `foreach`

```
static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}
```

Instrucción `break`

```
static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}
```

Instrucción `continue`

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}
```

Instrucción `goto`

```
static void Main(string[] args) {
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length) goto loop;
}
```

Instrucción `return`

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

Instrucción `yield`

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}
```

`throw` instrucciones `try` y

```
static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}
```

`checked` instrucciones `unchecked` y

```
static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);        // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);        // Overflow
    }
}
```

Instrucción `lock`

```
class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

Instrucción `using`

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

Clases y objetos

Las ***clases*** son los tipos más fundamentales de C#. Una clase es una estructura de datos que combina estados (campos) y acciones (métodos y otros miembros de función) en una sola unidad. Una clase proporciona una definición para ***instancias*** creadas dinámicamente de la clase, también conocidas como ***objetos***. Las clases admiten ***herencia*** y ***polimorfismo***, mecanismos por los que las ***clases derivadas*** pueden extender y especializar ***clases base***.

Las clases nuevas se crean mediante declaraciones de clase. Una declaración de clase se inicia con un encabezado que especifica los atributos y modificadores de la clase, el nombre de la clase, la clase base (si se indica) y las interfaces implementadas por la clase. Al encabezado le sigue el cuerpo de la clase, que consta de una lista de declaraciones de miembros escritas entre los delimitadores `{` y `}`.

La siguiente es una declaración de una clase simple denominada `Point`:

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Las instancias de clases se crean mediante el operador `new`, que asigna memoria para una nueva instancia, invoca un constructor para inicializar la instancia y devuelve una referencia a la instancia. Las instrucciones siguientes crean dos `Point` objetos y almacenan las referencias a esos objetos en dos variables:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

La memoria ocupada por un objeto se recupera automáticamente cuando el objeto ya no está en uso. No es necesario ni posible desasignar explícitamente objetos en C#.

Miembros

Los miembros de una clase son ***miembros estáticos*** o ***miembros de instancia***. Los miembros estáticos pertenecen a clases y los miembros de instancia pertenecen a objetos (instancias de clases).

En la tabla siguiente se proporciona información general sobre los tipos de miembros que puede contener una clase.

MIEMBRO	DESCRIPCIÓN
Constantes	Valores constantes asociados a la clase
Campos	Variables de la clase

MIEMBRO	DESCRIPCIÓN
Métodos	Cálculos y acciones que pueden realizarse mediante la clase
Propiedades	Acciones asociadas a la lectura y escritura de propiedades con nombre de la clase
Indizadores	Acciones asociadas a la indexación de instancias de la clase como una matriz
Eventos	Notificaciones que puede generar la clase
Operadores	Conversiones y operadores de expresión admitidos por la clase
Constructores	Acciones necesarias para inicializar instancias de la clase o la clase propiamente dicha
Destructores	Acciones que deben realizarse antes de que las instancias de la clase se descarten de forma permanente
Tipos	Tipos anidados declarados por la clase

Accesibilidad

Cada miembro de una clase tiene asociada una accesibilidad, que controla las regiones del texto del programa que pueden tener acceso al miembro. Existen cinco formas posibles de accesibilidad. Se resumen en la tabla siguiente.

ACCESIBILIDAD	AL
`public`	Acceso no limitado
`protected`	Acceso limitado a esta clase o a las clases derivadas de esta clase
`internal`	Acceso limitado a este programa
`protected internal`	Acceso limitado a este programa o a las clases derivadas de esta clase
`private`	Acceso limitado a esta clase

Parámetros de tipo

Una definición de clase puede especificar un conjunto de parámetros de tipo poniendo tras el nombre de clase una lista de nombres de parámetro de tipo entre corchetes angulares. Los parámetros de tipo se pueden usar en el cuerpo de las declaraciones de clase para definir los miembros de la clase. En el ejemplo siguiente, los parámetros de tipo de `Pair` son `TFirst`

y `TSecond`:

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

Un tipo de clase que se declara para tomar parámetros de tipo se denomina tipo de clase genérico. Los tipos struct, interfaz y delegado también pueden ser genéricos.

Cuando se usa la clase genérica, se deben proporcionar argumentos de tipo para cada uno de los parámetros de tipo:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two"
};
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string
```

Un tipo genérico con argumentos de tipo proporcionados `Pair<int,string>`, como el anterior, se denomina tipo construido.

Clases base

Una declaración de clase puede especificar una clase base colocando después del nombre de clase y los parámetros de tipo dos puntos seguidos del nombre de la clase base. Omitir una especificación de la clase base es igual que derivarla del tipo `object`. En el ejemplo siguiente, la clase base de `Point3D` es `Point` y la clase base de `Point` es `object`:

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

Una clase hereda a los miembros de su clase base. La herencia significa que una clase contiene implícitamente todos los miembros de su clase base, excepto la instancia y constructores estáticos, y los destructores de la clase base. Una clase derivada puede agregar nuevos miembros a aquellos de los que hereda, pero no puede quitar la definición de un miembro heredado. En el ejemplo anterior, `Point3D` hereda los campos `x` y `y` de `Point` y cada instancia de `Point3D` contiene tres campos: `x`, `y` y `z`.

Existe una conversión implícita de un tipo de clase a cualquiera de sus tipos de clase base. Por lo tanto, una variable de un tipo de clase puede hacer referencia a una instancia de esa clase o a una instancia de cualquier clase derivada. Por ejemplo, dadas las declaraciones de clase anteriores, una variable de tipo `Point` puede hacer referencia a una instancia de `Point` o `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Campos

Un campo es una variable que está asociada a una clase o a una instancia de una clase.

Un campo declarado con el `static` modificador define un ***campo estático***. Un campo estático identifica exactamente una ubicación de almacenamiento. Independientemente del número de instancias de una clase que se creen, siempre solo hay una copia de un campo estático.

Un campo declarado sin el `static` modificador define un ***campo de instancia***. Cada instancia de una clase contiene una copia independiente de todos los campos de instancia de esa clase.

En el ejemplo siguiente, cada instancia de la clase `Color` tiene una copia independiente de los campos de instancia `r`, `g` y `b`, pero solo hay una copia de los campos estáticos `Black`, `White`, `Red`, `Green` y `Blue`:

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

Como se muestra en el ejemplo anterior, los ***campos de solo lectura*** se puede declarar con un modificador `readonly`. La asignación a `readonly` un campo solo se puede producir como parte de la declaración del campo o en un constructor de la misma clase.

Métodos

Un ***método*** es un miembro que implementa un cálculo o una acción que puede realizar un objeto o una clase. A los ***métodos estáticos*** se accede a través de la clase. A los ***métodos de instancia*** se accede a través de instancias de la clase.

Los métodos tienen una lista (posiblemente vacía) de ***parámetros***, que representan valores o referencias de variables que se pasan al método, y un

tipo de valor ***devuelto***, que especifica el tipo del valor calculado y devuelto por el método. El tipo de valor devuelto `void` de un método es si no devuelve un valor.

Al igual que los tipos, los métodos también pueden tener un conjunto de parámetros de tipo, para lo cuales se deben especificar argumentos de tipo cuando se llama al método. A diferencia de los tipos, los argumentos de tipo a menudo se pueden deducir de los argumentos de una llamada al método y no es necesario proporcionarlos explícitamente.

La ***signatura*** de un método debe ser única en la clase en la que se declara el método. La signatura de un método se compone del nombre del método, el número de parámetros de tipo y el número, los modificadores y los tipos de sus parámetros. La signatura de un método no incluye el tipo de valor devuelto.

Parámetros

Los parámetros se usan para pasar valores o referencias a variables a métodos. Los parámetros de un método obtienen sus valores reales de los ***argumentos*** que se especifican cuando se invoca el método. Hay cuatro tipos de parámetros: parámetros de valor, parámetros de referencia, parámetros de salida y matrices de parámetros.

Un ***parámetro de valor*** se usa para el paso de parámetros de entrada. Un parámetro de valor corresponde a una variable local que obtiene su valor inicial del argumento que se ha pasado para el parámetro. Las modificaciones en un parámetro de valor no afectan el argumento que se pasa para el parámetro.

Los parámetros de valor pueden ser opcionales; se especifica un valor predeterminado para que se puedan omitir los argumentos correspondientes.

Un ***parámetro de referencia*** se usa para el paso de parámetros de entrada y salida. El argumento pasado para un parámetro de referencia debe ser una variable, y durante la ejecución del método, el parámetro de referencia representa la misma ubicación de almacenamiento que la variable del argumento. Un parámetro de referencia se declara con el modificador `ref`. En el ejemplo siguiente se muestra el uso de parámetros `ref`.

```
using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);            // Outputs "2 1"
    }
}
```

Un ***parámetro de salida*** se usa para pasar parámetros de salida. Un parámetro de salida es similar a un parámetro de referencia, salvo que el valor inicial del argumento proporcionado por el llamador no es importante. Un parámetro de salida se declara con el modificador `out`. En el ejemplo siguiente se muestra el uso de parámetros `out`.

```
using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder)
{
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem);    // Outputs "3 1"
    }
}
```

Una ***matriz de parámetros*** permite que se pasen a un método un número variable de argumentos. Una matriz de parámetros se declara con el modificador `params`. Solo el último parámetro de un método puede ser una matriz de parámetros y el tipo de una matriz de parámetros debe ser un tipo de matriz unidimensional. Los `Write` métodos `WriteLine` y de la `System.Console` clase son buenos ejemplos de uso de la matriz de parámetros. Se declaran de la manera siguiente.

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}
```

Dentro de un método que usa una matriz de parámetros, la matriz de parámetros se comporta exactamente igual que un parámetro normal de un tipo de matriz. Sin embargo, en una invocación de un método con una matriz de parámetros, es posible pasar un único argumento del tipo de matriz de parámetros o cualquier número de argumentos del tipo de elemento de la matriz de parámetros. En este caso, una instancia de matriz se e inicializa automáticamente con los argumentos dados. Este ejemplo

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

es equivalente a escribir lo siguiente.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Cuerpo del método y variables locales

El cuerpo de un método especifica las instrucciones que se ejecutan cuando se invoca el método.

Un cuerpo del método puede declarar variables que son específicas de la invocación del método. Estas variables se denominan ***variables locales***. Una declaración de variable local especifica un nombre de tipo, un nombre de variable y, posiblemente, un valor inicial. En el ejemplo siguiente se declara una variable local `i` con un valor inicial de cero y una variable local `j` sin ningún valor inicial.

```
using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}
```

C# requiere que se ***asigne definitivamente*** una variable local antes de que se pueda obtener su valor. Por ejemplo, si la declaración de `i` anterior no incluyera un valor inicial, el compilador notificaría un error con los usos posteriores de `i` porque `i` no se asignaría definitivamente en esos puntos del programa.

Puede usar una instrucción `return` para devolver el control a su llamador. En un método que devuelve `void`, las instrucciones `return` no pueden especificar una expresión. En un método que devuelve instrucciones `void` no `return`, debe incluir una expresión que calcule el valor devuelto.

Métodos estáticos y de instancia

Un método declarado con un `static` modificador es un ***método estático***. Un método estático no opera en una instancia específica y solo puede acceder directamente a miembros estáticos.

Un método declarado sin un `static` modificador es un ***método de instancia***. Un método de instancia opera en una instancia específica y puede acceder a miembros estáticos y de instancia. Se puede acceder explícitamente a la instancia en la que se invoca un método de instancia como `this`. Es un error hacer referencia a `this` en un método estático.

La siguiente clase `Entity` tiene miembros estáticos y de instancia.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

Cada instancia `Entity` contiene un número de serie (y probablemente alguna otra información que no se muestra aquí). El constructor `Entity` (que es como un método de instancia) inicializa la nueva instancia con el siguiente número de serie disponible. Dado que el constructor es un miembro de instancia, se le permite acceder al campo de instancia `serialNo` y al campo estático `nextSerialNo`.

Los métodos estáticos `GetNextSerialNo` y `SetNextSerialNo` pueden acceder al campo estático `nextSerialNo`, pero sería un error para ellas acceder directamente al campo de instancia `serialNo`.

En el ejemplo siguiente se muestra el uso `Entity` de la clase.

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

Tenga en cuenta que los métodos estáticos `SetNextSerialNo` y `GetNextSerialNo` se invocan en la clase, mientras que el método de instancia `GetSerialNo` se invoca en instancias de la clase.

Métodos virtual, de reemplazo y abstracto

Cuando una declaración de método de instancia incluye un modificador `virtual`, se dice que el método es un ***método virtual***. Cuando no `virtual` existe ningún modificador, se dice que el método es un ***método no virtual***.

Cuando se invoca un método virtual, el ***tipo en tiempo de ejecución*** de

la instancia para la que tiene lugar esa invocación determina la implementación del método real que se invocará. En una invocación de método no virtual, el ***tipo en tiempo de compilación*** de la instancia es el factor determinante.

Un método virtual puede ser ***reemplazado*** en una clase derivada. Cuando una declaración de método de instancia `override` incluye un modificador, el método invalida un método virtual heredado con la misma firma. Mientras que una declaración de método virtual introduce un método nuevo, una declaración de método de reemplazo especializa un método virtual heredado existente proporcionando una nueva implementación de ese método.

Un método ***abstracto*** es un método virtual sin implementación. Un método abstracto se declara con `abstract` el modificador y solo se permite en una clase que también se `abstract` declara. Un método abstracto debe reemplazarse en todas las clases derivadas no abstractas.

En el ejemplo siguiente se declara una clase abstracta, `Expression`, que representa un nodo de árbol de expresión y tres clases derivadas, `Constant`, `VariableReference` y `Operation`, que implementan nodos de árbol de expresión para constantes, referencias a variables y operaciones aritméticas. (Esto es similar a, pero no debe confundirse con los tipos de árbol de expresión introducidos en los tipos de árbol de expresión).

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```

Las cuatro clases anteriores se pueden usar para modelar expresiones aritméticas. Por ejemplo, usando instancias de estas clases, la expresión `x + 3` se puede representar de la manera siguiente.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

El método `Evaluate` de una instancia `Expression` se invoca para evaluar la expresión determinada y generar un valor `double`. El método toma como argumento un `Hashtable` que contiene nombres de variables (como claves de las entradas) y valores (como valores de las entradas). El `Evaluate` método es un método abstracto virtual, lo que significa que las clases derivadas no abstractas deben invalidarlo para proporcionar una implementación real.

Una implementación de `Constant` de `Evaluate` simplemente devuelve la constante almacenada. Una `VariableReference` implementación de busca el nombre de la variable en la tabla hash y devuelve el valor resultante. Una implementación de `Operation` evalúa primero los operandos izquierdo y derecho (mediante la invocación recursiva de sus métodos `Evaluate`) y luego realiza la operación aritmética correspondiente.

El siguiente programa usa las clases `Expression` para evaluar la expresión `x * (y + 2)` para los distintos valores de `x` y `y`.

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));        // Outputs "16.5"
    }
}
```

Sobrecarga de métodos

La ***sobrecarga*** de métodos permite que varios métodos de la misma clase tengan el mismo nombre mientras tengan signaturas únicas. Al compilar una invocación de un método sobrecargado, el compilador usa la ***resolución de sobrecarga*** para determinar el método concreto que de invocará. La resolución de sobrecarga busca el método que mejor coincida con los argumentos o informa de un error si no se puede encontrar ninguna mejor coincidencia. En el ejemplo siguiente se muestra la resolución de sobrecarga en vigor. El comentario para cada invocación del

método `Main` muestra qué método se invoca realmente.

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                // Invokes F()
        F(1);               // Invokes F(int)
        F(1.0);             // Invokes F(double)
        F("abc");           // Invokes F(object)
        F((double)1);       // Invokes F(double)
        F((object)1);       // Invokes F(object)
        F<int>(1);          // Invokes F<T>(T)
        F(1, 1);            // Invokes F(double, double)
    }
}
```

Tal como se muestra en el ejemplo, un método determinado siempre se puede seleccionar mediante la conversión explícita de los argumentos en los tipos de parámetros exactos o el suministro explícito de los argumentos de tipo.

Otros miembros de función

Los miembros que contienen código ejecutable se conocen colectivamente como ***miembros de función*** de una clase. En la sección anterior se han descrito métodos, que son el tipo principal de los miembros de función. En esta sección se describen los demás tipos de miembros de C#función admitidos por: constructores, propiedades, indexadores, eventos, operadores y destructores.

En el código siguiente se muestra una clase `List<T>` genérica denominada, que implementa una lista de objetos que se va a ampliar. La clase contiene varios ejemplos de los tipos más comunes de miembros de función.

```
public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
```

```
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...
    public int Count {
        get { return count; }
    }
    public int Capacity {
        get {
            return items.Length;
        }
        set {
            if (value < count) value = count;
            if (value != items.Length) {
                T[] newItems = new T[value];
                Array.Copy(items, 0, newItems, 0, count);
                items = newItems;
            }
        }
    }

    // Indexer...
    public T this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods...
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
        return true;
    }

    // Event...
    public event EventHandler Changed;

    // Operators...
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
    public static bool operator !=(List<T> a, List<T> b) {
```

```
        return !Equals(a, b);
    }
}
```

Constructores

C# admite constructores de instancia y estáticos. Un ***constructor de instancia*** es un miembro que implementa las acciones necesarias para inicializar una instancia de una clase. Un ***constructor estático*** es un miembro que implementa las acciones necesarias para inicializar una clase en sí misma cuando se carga por primera vez.

Un constructor se declara como un método sin ningún tipo de valor devuelto y el mismo nombre que la clase contenedora. Si una declaración de constructor incluye `static` un modificador, declara un constructor estático. De lo contrario, declara un constructor de instancia.

Los constructores de instancias se pueden sobrecargar. Por ejemplo, la clase `List<T>` declara dos constructores de instancia, una sin parámetros y otra que toma un parámetro `int`. Los constructores de instancia se invocan mediante el operador `new`. Las instrucciones siguientes asignan `List<string>` dos instancias de mediante cada uno de los constructores de la `List` clase.

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

A diferencia de otros miembros, los constructores de instancia no se heredan y una clase no tiene ningún constructor de instancia que no sea el que se declara realmente en la clase. Si no se proporciona ningún constructor de instancia para una clase, se proporciona automáticamente uno vacío sin ningún parámetro.

Propiedades

Las ***propiedades*** son una extensión natural de los campos. Ambos son miembros con nombre con tipos asociados y la sintaxis para acceder a los campos y las propiedades es la misma. Sin embargo, a diferencia de los campos, las propiedades no denotan ubicaciones de almacenamiento. Las propiedades tienen ***descriptores de acceso*** que especifican las instrucciones que se ejecutan cuando se leen o escriben sus valores.

Una propiedad se declara como un campo, salvo que la declaración finaliza con un `get` descriptor de acceso `set` o un descriptor de acceso `{` escrito `}` entre los delimitadores y en lugar de terminar en un punto y coma. Una propiedad que tiene tanto un `get` descriptor `set` de acceso como un descriptor de acceso es una propiedad ***de lectura y escritura***, una propiedad que solo tiene un `get` descriptor de acceso es una ***propiedad de solo lectura***y una propiedad que solo tiene un `set` descriptor de acceso es un ***propiedad de solo escritura***.

Un `get` descriptor de acceso corresponde a un método sin parámetros con un valor devuelto del tipo de propiedad. Excepto como destino de una asignación, cuando se hace referencia a una propiedad en una expresión, el `get` descriptor de acceso de la propiedad se invoca para calcular el valor de la propiedad.

Un `set` descriptor de acceso corresponde a un método con `value` un solo parámetro denominado y ningún tipo de valor devuelto. Cuando se hace referencia a una propiedad como el destino de una asignación o como el operando `--` de o `set` , el descriptor de `++` acceso se invoca con un argumento que proporciona el nuevo valor.

La clase `List<T>` declara dos propiedades, `Count` y `Capacity`, que son de solo lectura y de lectura y escritura, respectivamente. El siguiente es un ejemplo de uso de estas propiedades.

```
List<string> names = new List<string>();
names.Capacity = 100;            // Invokes set accessor
int i = names.Count;             // Invokes get accessor
int j = names.Capacity;          // Invokes get accessor
```

De forma similar a los campos y métodos, C# admite propiedades de instancia y propiedades estáticas. Las propiedades estáticas se declaran con el `static` modificador y las propiedades de instancia se declaran sin ella.

Los descriptores de acceso de una propiedad pueden ser virtuales. Cuando una declaración de propiedad incluye un modificador `virtual`, `abstract` o `override`, se aplica a los descriptores de acceso de la propiedad.

Indizadores

Un ***indexador*** es un miembro que permite indexar de la misma manera que una matriz. Un indexador se declara como una propiedad, excepto por el hecho que el nombre del miembro es `this`, seguido por una lista de parámetros que se escriben entre los delimitadores `[` y `]`. Los parámetros están disponibles en los descriptores de acceso del indexador. De forma similar a las propiedades, los indexadores pueden ser lectura y escritura, de solo lectura y de solo escritura, y los descriptores de acceso de un indexador pueden ser virtuales.

La clase `List` declara un único indexador de lectura y escritura que toma un parámetro `int`. El indexador permite indexar instancias de `List` con valores `int`. Por ejemplo

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Los indexadores se pueden sobrecargar, lo que significa que una clase puede declarar varios indexadores siempre y cuando el número o los tipos de sus parámetros sean diferentes.

Eventos

Un ***evento*** es un miembro que permite que una clase u objeto proporcionen notificaciones. Un evento se declara como un campo salvo que la Declaración incluye una `event` palabra clave y el tipo debe ser un tipo de delegado.

Dentro de una clase que declara un miembro de evento, el evento se comporta como un campo de un tipo delegado (siempre que el evento no sea abstracto y no declare descriptores de acceso). El campo almacena una referencia a un delegado que representa los controladores de eventos que se han agregado al evento. Si no hay ningún controlador de eventos presente, el `null` campo es.

La clase `List<T>` declara un único miembro de evento llamado `Changed`, lo que indica que se ha agregado un nuevo elemento a la lista. El evento lo desencadena el `OnChanged` método virtual, que primero comprueba si el evento es `null` (lo que significa que no hay ningún controlador presente). `Changed` La noción de generar un evento es equivalente exactamente a invocar el delegado representado por el evento; por lo tanto, no hay ninguna construcción especial de lenguaje para generar eventos.

Los clientes reaccionan a los eventos mediante ***controladores de eventos***. Los controladores de eventos se asocian mediante el operador `+=` y se quitan con el operador `-=`. En el ejemplo siguiente se asocia un controlador de eventos con el evento `Changed` de un objeto `List<string>`.

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);        // Outputs "3"
    }
}
```

Para escenarios avanzados donde se desea controlar el almacenamiento subyacente de un evento, una declaración de evento puede proporcionar explícitamente los descriptores de acceso `add` y `remove`, que son similares en cierto modo al descriptor de acceso `set` de una propiedad.

Operadores

Un *operador* es un miembro que define el significado de aplicar un operador de expresión determinado a las instancias de una clase. Se pueden definir tres tipos de operadores: operadores unarios, operadores binarios y operadores de conversión. Todos los operadores se deben declarar como `public` y `static`.

La clase `List<T>` declara dos operadores, `operator==` y `operator!=`, y de este modo proporciona un nuevo significado a expresiones que aplican esos operadores a instancias `List`. En concreto, los operadores definen la igualdad `List<T>` de dos instancias como la comparación de cada uno de los `Equals` objetos contenidos mediante sus métodos. En el ejemplo

siguiente se usa el operador `==` para comparar dos instancias `List<int>`.

```
using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);        // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);        // Outputs "False"
    }
}
```

El primer objeto `Console.WriteLine` genera `True` porque las dos listas contienen el mismo número de objetos con los mismos valores en el mismo orden. Si `List<T>` no hubiera definido `operator==`, el primer objeto `Console.WriteLine` habría generado `False` porque `a` y `b` hacen referencia a diferentes instancias de `List<int>`.

Destructores

Un ***destructor*** es un miembro que implementa las acciones necesarias para destruir una instancia de una clase. Los destructores no pueden tener parámetros, no pueden tener modificadores de accesibilidad y no se pueden invocar explícitamente. El destructor de una instancia se invoca automáticamente durante la recolección de elementos no utilizados.

El recolector de elementos no utilizados tiene una latitud ancha a la hora de decidir cuándo recopilar objetos y ejecutar destructores. En concreto, el tiempo de las invocaciones de destructor no es determinista y los destructores se pueden ejecutar en cualquier subproceso. Por estas y otras razones, las clases deben implementar destructores solo cuando no sean factibles otras soluciones.

La instrucción `using` proporciona un mejor enfoque para la destrucción de objetos.

Estructuras

Al igual que las clases, los ***structs*** son estructuras de datos que pueden contener miembros de datos y miembros de función, pero a diferencia de las clases, los structs son tipos de valor y no requieren asignación del montón. Una variable de un tipo de struct almacena directamente los datos del struct, mientras que una variable de un tipo de clase almacena una referencia a un objeto asignado dinámicamente. Los tipos struct no admiten la herencia especificada por el usuario y todos los tipos de struct se heredan implícitamente del tipo `object`.

Los structs son particularmente útiles para estructuras de datos pequeñas que tengan semánticas de valor. Los números complejos, los puntos de un sistema de coordenadas o los pares clave-valor de un diccionario son buenos ejemplos de structs. El uso de un struct en lugar de una clase para

estructuras de datos pequeñas puede suponer una diferencia sustancial en el número de asignaciones de memoria que realiza una aplicación. Por ejemplo, el siguiente programa crea e inicializa una matriz de 100 puntos. Si `Point` se implementa como una clase, se crean instancias de 101 objetos distintos: uno para la matriz y uno por cada uno de los 100 elementos.

```
class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

Una alternativa es crear `Point` un struct.

```
struct Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Ahora, se crea la instancia de un solo objeto: la de la matriz, y las instancias de `Point` se asignan en línea dentro de la matriz.

Los structs se invocan con el operador `new`, pero eso no implica que se asigne memoria. En lugar de asignar dinámicamente un objeto y devolver una referencia a él, un constructor de structs simplemente devuelve el valor del struct propiamente dicho (normalmente en una ubicación temporal en la pila) y este valor se copia luego cuando es necesario.

Con las clases, es posible que dos variables hagan referencia al mismo objeto y, que por tanto, las operaciones en una variable afecten al objeto al que hace referencia la otra variable. Con los struct, cada variable tiene su propia copia de los datos y no es posible que las operaciones en una afecten a la otra. Por ejemplo, la salida generada por el fragmento de código siguiente depende `Point` de si es una clase o un struct.

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

Si `Point` es una clase, el resultado es `20` porque `a` y `b` hacen referencia al mismo objeto. Si `Point` es un struct, el resultado es `10` porque la asignación de `a` a `b` crea una copia del valor, y esta copia no se ve afectada por la asignación subsiguiente a `a.x`.

En el ejemplo anterior se resaltan dos de las limitaciones de los structs. En primer lugar, copiar un struct entero normalmente es menos eficaz que copiar una referencia a un objeto, por lo que el paso de parámetros de asignación y valor puede ser más costoso con structs que con tipos de referencia. En segundo lugar, a excepción de los parámetros `ref` y `out`, no es posible crear referencias a structs, que excluyen su uso en varias situaciones.

Matrices

Una ***matriz*** es una estructura de datos que contiene un número de variables a las que se accede mediante índices calculados. Las variables contenidas en una matriz, denominadas también ***elementos*** de la matriz, son todas del mismo tipo y este tipo se conoce como ***tipo de elemento*** de la matriz.

Los tipos de matriz son tipos de referencia, y la declaración de una variable de matriz simplemente establece un espacio reservado para una referencia a una instancia de matriz. Las instancias de matriz reales se crean dinámicamente en tiempo de ejecución `new` mediante el operador. La `new` operación especifica la ***longitud*** de la nueva instancia de la matriz, que se corrige para la duración de la instancia. Los índices de los elementos de una matriz van de `0` a `Length - 1`. El operador `new` inicializa automáticamente los elementos de una matriz a su valor predeterminado, que, por ejemplo, es cero para todos los tipos numéricos y `null` para todos los tipos de referencias.

En el ejemplo siguiente se crea una matriz de elementos `int`, se inicializa la matriz y se imprime el contenido de la matriz.

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

Este ejemplo se crea y se pone en funcionamiento en una ***matriz unidimensional***. C# también admite ***matrices multidimensionales***. El número de dimensiones de un tipo de matriz, conocido también como ***rango*** del tipo de matriz, es una más el número de comas escritas entre los corchetes del tipo de matriz. En el ejemplo siguiente se asigna una matriz unidimensional, bidimensional y de tres dimensiones.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

La matriz `a1` contiene 10 elementos, la matriz `a2` 50 (10 × 5) elementos y la matriz `a3` 100 (10 × 5 × 2) elementos.

El tipo de elemento de una matriz puede ser cualquiera, incluido un tipo de matriz. Una matriz con elementos de un tipo de matriz a veces se conoce como ***matriz escalonada*** porque las longitudes de las matrices de elementos no tienen que ser iguales. En el ejemplo siguiente se asigna una matriz de matrices de `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

La primera línea crea una matriz con tres elementos, cada uno de tipo `int[]` y cada uno con un valor inicial de `null`. Las líneas posteriores inicializan entonces los tres elementos con referencias a instancias de matriz individuales de longitud variable.

El `new` operador permite especificar los valores iniciales de los elementos de la matriz mediante un ***inicializador de matriz***, que es una lista de expresiones escritas entre `{` los delimitadores y. `}` En el ejemplo siguiente se asigna e inicializa un tipo `int[]` con tres elementos.

```
int[] a = new int[] {1, 2, 3};
```

Tenga en cuenta que la longitud de la matriz se deduce del número de expresiones `{` entre `}` y. Las declaraciones de variable local y campo se pueden acortar más para que así no sea necesario reformular el tipo de matriz.

```
int[] a = {1, 2, 3};
```

Los dos ejemplos anteriores son equivalentes a lo siguiente:

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

Interfaces

Una ***interfaz*** define un contrato que se puede implementar mediante clases y structs. Una interfaz puede contener métodos, propiedades, eventos e indexadores. Una interfaz no proporciona implementaciones de los miembros que define, simplemente especifica los miembros que se deben proporcionar mediante clases o structs que implementan la interfaz.

Las interfaces pueden usar ***herencia múltiple***. En el ejemplo siguiente, la interfaz `IComboBox` hereda de `ITextBox` y `IListBox`.

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

Las clases y los structs pueden implementar varias interfaces. En el ejemplo siguiente, la clase `EditBox` implementa `IControl` y `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

Cuando una clase o un struct implementan una interfaz determinada, las instancias de esa clase o struct se pueden convertir implícitamente a ese tipo de interfaz. Por ejemplo

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

En casos donde una instancia no se conoce estáticamente para implementar una interfaz determinada, se pueden usar conversiones de tipo dinámico. Por ejemplo, las siguientes instrucciones usan conversiones de tipo dinámico para obtener las implementaciones `IDataBound` de un objeto y de la `IControl` interfaz. Dado que el tipo real del objeto es `EditBox`, las conversiones se realizan correctamente.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

En la clase `EditBox` anterior, el `Paint` método de la `IControl` interfaz y el `Bind` método de la `IDataBound` interfaz se implementan `public` mediante miembros. C#también admite ***implementaciones explícitas de miembros de interfaz***, con las que la clase o estructura puede `public`

evitar la creación de miembros. Una implementación de miembro de interfaz explícito se escribe con el nombre de miembro de interfaz completo. Por ejemplo, la clase `EditBox` podría implementar los métodos `IControl.Paint` y `IDataBound.Bind` mediante implementaciones de miembros de interfaz explícitos del modo siguiente.

```
public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}
```

Solo se puede acceder a los miembros de interfaz explícitos mediante el tipo de interfaz. Por ejemplo, la implementación de `IControl.Paint` proporcionada por la clase `EditBox` anterior solo se puede invocar `IControl` convirtiendo primero la `EditBox` referencia al tipo de interfaz.

```
EditBox editBox = new EditBox();
editBox.Paint();                        // Error, no such method
IControl control = editBox;
control.Paint();                        // Ok
```

Enumeraciones

Un ***tipo de enumeración*** es un tipo de valor distinto con un conjunto de constantes con nombre. En el ejemplo siguiente se declara y se utiliza un tipo `Color` de enumeración denominado con `Red` tres `Green` valores constantes `Blue`,, y.

```
using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

Cada tipo de enumeración tiene un tipo entero correspondiente denominado el tipo ***subyacente*** del tipo de enumeración. Un tipo de enumeración que no declara explícitamente un tipo subyacente tiene un tipo `int` subyacente de. El formato de almacenamiento y el intervalo de valores posibles de un tipo enum vienen determinados por su tipo subyacente. El conjunto de valores que puede tomar un tipo de enumeración no está limitado por sus miembros de enumeración. En concreto, cualquier valor del tipo subyacente de una enumeración se puede convertir al tipo de enumeración y es un valor válido distinto de ese tipo de enumeración.

En el ejemplo siguiente se declara un tipo de `Alignment` enumeración denominado con un `sbyte` tipo subyacente de.

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

Como se muestra en el ejemplo anterior, una declaración de miembro de enumeración puede incluir una expresión constante que especifique el valor del miembro. El valor constante para cada miembro de la enumeración debe estar en el intervalo del tipo subyacente de la

enumeración. Cuando una declaración de miembro de enumeración no especifica explícitamente un valor, el miembro recibe el valor cero (si es el primer miembro del tipo de enumeración) o el valor del miembro de enumeración anterior textual más uno.

Los valores de enumeración se pueden convertir en valores enteros y viceversa mediante conversiones de tipos. Por ejemplo

```
int i = (int)Color.Blue;        // int i = 2;
Color c = (Color)2;             // Color c = Color.Blue;
```

El valor predeterminado de cualquier tipo de enumeración es el valor entero cero convertido al tipo de enumeración. En los casos en los que las variables se inicializan automáticamente en un valor predeterminado, este es el valor dado a las variables de los tipos de enumeración. Para que el valor predeterminado de un tipo de enumeración sea fácilmente disponible, el literal `0` se convierte implícitamente a cualquier tipo de enumeración. Por tanto, el siguiente código es válido.

```
Color c = 0;
```

Delegados

Un ***tipo de delegado*** representa las referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Los delegados permiten tratar métodos como entidades que se puedan asignar a variables y se puedan pasar como parámetros. Los delegados son similares al concepto de punteros de función en otros lenguajes, pero a diferencia de los punteros de función, los delegados están orientados a objetos y presentan seguridad de tipos.

En el siguiente ejemplo se declara y usa un tipo de delegado denominado `Function`.

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles =  Apply(a, m.Multiply);
    }
}
```

Una instancia del tipo de delegado `Function` puede hacer referencia a cualquier método que tome un argumento `double` y devuelva un valor `double`. El `Apply` método aplica un determinado `Function` a los elementos de un `double[]`, devolviendo `double[]` un con los resultados. En el método `Main`, `Apply` se usa para aplicar tres funciones diferentes a un valor `double[]`.

Un delegado puede hacer referencia a un método estático (como `Square` o `Math.Sin` en el ejemplo anterior) o un método de instancia (como `m.Multiply` en el ejemplo anterior). Un delegado que hace referencia a un método de instancia también hace referencia a un objeto determinado y, cuando se invoca el método de instancia a través del delegado, ese objeto se convierte en `this` en la invocación.

Los delegados también pueden crearse mediante funciones anónimas, que son "métodos insertados" que se crean sobre la marcha. Las funciones anónimas pueden ver las variables locales de los métodos adyacentes. Por lo tanto, el ejemplo de multiplicador anterior se puede escribir más fácilmente `Multiplier` sin usar una clase:

```
double[] doubles =  Apply(a, (double x) => x * 2.0);
```

Una propiedad interesante y útil de un delegado es que no sabe ni necesita conocer la clase del método al que hace referencia; lo único que importa es que el método al que se hace referencia tenga los mismos parámetros y el tipo de valor devuelto que el delegado.

Atributos

Los tipos, los miembros y otras entidades en un programa de C # admiten modificadores que controlan ciertos aspectos de su comportamiento. Por ejemplo, la accesibilidad de un método se controla mediante los modificadores `public`, `protected`, `internal` y `private`. C # generaliza esta funcionalidad de manera que los tipos de información declarativa definidos por el usuario se puedan adjuntar a las entidades del programa y recuperarse en tiempo de ejecución. Los programas especifican esta información declarativa adicional mediante la definición y el uso de ***atributos***.

En el ejemplo siguiente se declara un atributo `HelpAttribute` que se puede colocar en entidades de programa para proporcionar vínculos a la documentación asociada.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

Todas las clases de atributo derivan de la `System.Attribute` clase base proporcionada por el .NET Framework. Los atributos se pueden aplicar proporcionando su nombre, junto con cualquier argumento, entre corchetes, justo antes de la declaración asociada. Si el nombre de un atributo termina `Attribute` en, esa parte del nombre se puede omitir cuando se hace referencia al atributo. Por ejemplo, el atributo `HelpAttribute` se puede usar de la manera siguiente.

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic =
"Display")]
    public void Display(string text) {}
}
```

En este ejemplo se adjunta `HelpAttribute` un a `Widget` la `Display` clase y `HelpAttribute` otro al método en la clase. Los constructores públicos de una clase de atributos controlan la información que se debe proporcionar cuando el atributo se adjunta a una entidad de programa. Se puede proporcionar información adicional haciendo referencia a las propiedades públicas de lectura y escritura de la clase de atributos (como la referencia a la propiedad `Topic` usada anteriormente).

En el ejemplo siguiente se muestra cómo se puede recuperar la información de atributos de una entidad de programa determinada en tiempo de ejecución mediante la reflexión.

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine("  Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

Cuando se solicita un atributo determinado mediante reflexión, se invoca al constructor de la clase de atributos con la información proporcionada en el origen del programa y se devuelve la instancia de atributo resultante. Si se proporciona información adicional mediante propiedades, dichas propiedades se establecen en los valores dados antes de devolver la instancia del atributo.

Coincidencia de patrones C# para 7

29/06/2020 • 30 minutes to read • Edit Online

Las extensiones de coincidencia C# de patrones para permiten muchas de las ventajas de los tipos de datos algebraicos y la coincidencia de patrones de los lenguajes funcionales, pero de una manera que se integra sin problemas con la sensación del lenguaje subyacente. Las características básicas son: tipos de registros, que son tipos cuyo significado semántico se describe mediante la forma de los datos. y la coincidencia de patrones, que es un nuevo formulario de expresión que permite la descomposición de varios niveles de estos tipos de datos. Los elementos de este enfoque están inspirados en las características relacionadas en F# los lenguajes de programación y Scala.

Expresión is

El operador `is` se extiende para probar una expresión con un *patrón*.

```
relational_expression
    : relational_expression 'is' pattern
    ;
```

Esta forma de *relational_expression* se suma a los formularios existentes en la C# especificación. Es un error en tiempo de compilación si el *relational_expression* a la izquierda del token de `is` no designa un valor o no tiene un tipo.

Cada *identificador* del patrón introduce una nueva variable local que se *asigna definitivamente* después de que el operador de `is` sea `true` (es decir, se *asigna definitivamente cuando es true*).

> Nota: técnicamente existe una ambigüedad entre el *tipo* en un `is-expression` y *constant_pattern*, cualquiera de los cuales puede ser un análisis válido de un identificador calificado. Intentamos enlazarlo como un tipo para la compatibilidad con versiones anteriores del lenguaje; solo si se produce un error, se resuelve como hacemos en otros contextos, lo primero que se encontró (que debe ser una constante o un tipo). Esta ambigüedad solo está presente en la parte derecha de una expresión de `is`.

Patrones

Los patrones se utilizan en el operador `is` y en un *switch_statement* para expresar la forma de los datos con los que se van a comparar los datos entrantes. Los patrones pueden ser recursivos para que se puedan comparar partes de los datos con subpatrones.

```
pattern
    : declaration_pattern
    | constant_pattern
    | var_pattern
    ;

declaration_pattern
    : type simple_designation
    ;

constant_pattern
    : shift_expression
    ;

var_pattern
    : 'var' simple_designation
    ;
```

> Nota: técnicamente existe una ambigüedad entre el *tipo* en un `is-expression` y *constant_pattern*, cualquiera de los cuales puede ser un análisis válido de un identificador calificado. Intentamos enlazarlo como un tipo para la compatibilidad con versiones anteriores del lenguaje; solo si se produce un error, se resuelve como hacemos en otros contextos, lo primero que se encontró (que debe ser una constante o un tipo). Esta ambigüedad solo está presente en la parte derecha de una expresión de `is`.

Modelo de declaración

El *declaration_pattern* ambos comprueba que una expresión es de un tipo determinado y la convierte a ese tipo si la prueba se realiza correctamente. Si el *simple_designation* es un identificador, introduce una variable local del tipo especificado denominado por el identificador especificado. Esa variable local se *asigna definitivamente* cuando el resultado de la operación de coincidencia de patrones es true.

```
declaration_pattern
    : type simple_designation
    ;
```

La semántica en tiempo de ejecución de esta expresión es que prueba el tipo en tiempo de ejecución del operando del lado izquierdo *relational_expression* en el *tipo* del patrón. Si es de ese tipo en tiempo de ejecución (o algún subtipo), el resultado del `is operator` es `true`. Declara una nueva variable local denominada por el *identificador* al que se asigna el valor del operando izquierdo cuando el resultado es `true`.

Ciertas combinaciones de tipo estático del lado izquierdo y del tipo especificado se consideran incompatibles y producen un error en tiempo de compilación. Se dice que un valor de tipo estático `E` es *compatible* con el tipo `T` si existe una conversión de identidad, una conversión de referencia implícita, una conversión boxing, una conversión de referencia explícita o una conversión unboxing de `E` a `T`. Es un error en tiempo de compilación si una expresión de tipo `E` no es compatible con el tipo en un patrón de tipo con el que se encuentra una coincidencia.

> Nota: en C# 7,1 se amplía esto para permitir una operación de coincidencia de patrones si el tipo de entrada o el tipo `T` es un tipo abierto. Este párrafo se sustituye por lo siguiente:
>
> Ciertas combinaciones de tipo estático del lado izquierdo y del tipo especificado se consideran incompatibles y producen un error en tiempo de compilación. Se dice que un valor de tipo estático `E` es *compatible* con el tipo `T` si existe una conversión de identidad, una conversión de referencia implícita, una conversión boxing, una conversión de referencia explícita o una conversión unboxing de `E` a `T`, **o bien, si `E` o `T` es un tipo abierto**. Es un error en tiempo de compilación si una expresión de tipo `E` no es compatible con el tipo en un patrón de tipo con el que se encuentra una coincidencia.

El modelo de declaración es útil para realizar pruebas de tipo en tiempo de ejecución de tipos de referencia y reemplaza la expresión

```
var v = expr as Type;
if (v != null) { // code using v }
```

Con el poco más conciso

```
if (expr is Type v) { // code using v }
```

Es un error si el *tipo* es un tipo de valor que acepta valores NULL.

El modelo de declaración se puede usar para probar valores de tipos que aceptan valores NULL: un valor de tipo `Nullable<T>` (o un `T` con conversión boxing) coincide con un patrón de tipo `T2 id` si el valor no es NULL y el tipo de `T2` es `T`, o algún tipo base o interfaz de `T`. Por ejemplo, en el fragmento de código

```
int? x = 3;
if (x is int v) { // code using v }
```

La condición de la instrucción `if` es `true` en tiempo de ejecución y la variable `v` contiene el valor `3` de tipo `int` dentro del bloque.

Patrón de constante

```
constant_pattern
    : shift_expression
    ;
```

Un patrón de constante prueba el valor de una expresión con respecto a un valor constante. La constante puede ser cualquier expresión constante, como un literal, el nombre de una variable `const` declarada, una constante de enumeración o una expresión `typeof`.

Si tanto *e* como *c* son de tipos enteros, se considera que el patrón coincide si el resultado de la expresión `e == c` es `true`.

De lo contrario, se considera que el patrón coincide si `object.Equals(e, c)` devuelve `true`. En este caso, se trata de un error en tiempo de compilación si el tipo estático de *e* no es *compatible* con el tipo de la constante.

Patrón var

```
var_pattern
    : 'var' simple_designation
    ;
```

Una expresión *e* coincide con un *var_pattern* siempre. En otras palabras, una coincidencia con un *patrón var* siempre se realiza correctamente. Si el *simple_designation* es un identificador, en tiempo de ejecución el valor de *e* se enlaza a una variable local recién introducida. El tipo de la variable local es el tipo estático de *e*.

Es un error si el nombre `var` enlaza a un tipo.

Instrucción switch

La instrucción `switch` se extiende para seleccionar la ejecución del primer bloque que tiene un patrón asociado que coincide con la *expresión switch*.

```
switch_label
    : 'case' complex_pattern case_guard? ':'
    | 'case' constant_expression case_guard? ':'
    | 'default' ':'
    ;

case_guard
    : 'when' expression
    ;
```

No se define el orden en el que se buscan coincidencias con los patrones. Se permite que un compilador coincida con los patrones desordenados y reutilizar los resultados de los patrones ya coincidentes para calcular el resultado de la coincidencia de otros patrones.

Si hay una *protección de casos* , su expresión es de tipo `bool`. Se evalúa como una condición adicional que se debe satisfacer para que el caso se considere satisfecho.

Es un error si un *switch_label* puede no tener ningún efecto en el tiempo de ejecución porque su patrón está en los casos anteriores. [TODO: deberíamos ser más precisos sobre las técnicas que el compilador necesita usar para alcanzar esta resolución.]

Una variable de patrón declarada en un *switch_label* se asigna definitivamente en su bloque case si y solo si ese bloque Case contiene exactamente un *switch_label*.

[TODO: deberíamos especificar Cuándo se puede obtener acceso a un *bloque switch* .]

Ámbito de las variables de patrón

El ámbito de una variable declarada en un patrón es el siguiente:

- Si el patrón es una etiqueta de caso, el ámbito de la variable es el *bloque de casos*.

En caso contrario, la variable se declara en una expresión *is_pattern* y su ámbito se basa en la construcción que incluye inmediatamente la expresión que contiene la expresión *is_pattern* de la siguiente manera:

- Si la expresión se encuentra en una expresión lambda con cuerpo de expresión, su ámbito es el cuerpo de la lambda.
- Si la expresión se encuentra en un método o propiedad con forma de expresión, su ámbito es el cuerpo del método o propiedad.
- Si la expresión está en una cláusula `when` de una cláusula `catch`, su ámbito es la cláusula `catch`.
- Si la expresión se encuentra en un *iteration_statement*, su ámbito es simplemente esa instrucción.
- De lo contrario, si la expresión está en otra forma de instrucción, su ámbito es el ámbito que contiene la instrucción.

Con el fin de determinar el ámbito, se considera que una *embedded_statement* está en su propio ámbito. Por ejemplo, la gramática de una *if_statement* es

```
if_statement
    : 'if' '(' boolean_expression ')' embedded_statement
    | 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
    ;
```

Por tanto, si la instrucción controlada de un *if_statement* declara una variable de patrón, su ámbito está restringido a ese *embedded_statement*:

```
if (x) M(y is var z);
```

En este caso, el ámbito de `z` es la instrucción incrustada `M(y is var z);`.

Otros casos son errores por otros motivos (por ejemplo, en el valor predeterminado de un parámetro o en un atributo, ambos son un error porque esos contextos requieren una expresión constante).

> En C# 7,3 hemos agregado los siguientes contextos en los que se puede declarar una variable de patrón:
>
> - Si la expresión se encuentra en un *inicializador de constructor*, su ámbito es el *inicializador del constructor* y el cuerpo del constructor.
> - Si la expresión se encuentra en un inicializador de campo, su ámbito es el *equals_value_clause* en el que aparece.
> - Si la expresión se encuentra en una cláusula de consulta que se especifica para traducirse en el cuerpo de una expresión lambda, su ámbito es simplemente esa expresión.

Cambios en la desambiguación sintáctica

Existen situaciones en las que se incluyen los genéricos en los que la C# gramática es ambigua y en la especificación del lenguaje se indica cómo resolver esas ambigüedades:

> **7.6.5.2 ambigüedades de la gramática**
>
> Las producciones de *nombre simple* (§ 7.6.3) y *acceso a miembros* (§ 7.6.5) pueden dar lugar a ambigüedades en la gramática de expresiones. Por ejemplo, la instrucción:
>
> ```
> F(G<A,B>(7));
> ```
>
> podría interpretarse como una llamada a `F` con dos argumentos, `G < A` y `B > (7)`. Como alternativa, podría interpretarse como una llamada a `F` con un argumento, que es una llamada a un método genérico `G` con dos argumentos de tipo y un argumento normal.

> Si una secuencia de tokens se puede analizar (en contexto) como un *nombre simple* (§ 7.6.3), *acceso a miembro* (§ 7.6.5) o *puntero-miembro-acceso* (§ 18.5.2) que termina con una *lista de argumentos de tipo* (§ 4.4.1), se examina el token que sigue inmediatamente al cierre `>` token. Si es uno de
>
> ```
> ( ) ] } : ; , . ? == != | ^
> ```
>
> a continuación, la *lista de argumentos de tipo* se conserva como parte del acceso *simple*, de *acceso a miembros* o de miembro de *puntero* , y se descarta cualquier otro posible análisis de la secuencia de tokens. De lo contrario, la *lista de argumentos de tipo* no se considera parte del *nombre simple*, *acceso a miembros* o > puntero- *miembro-acceso*, incluso si no hay otro posible análisis de la secuencia de tokens. Tenga en cuenta que estas reglas no se aplican al analizar una *lista de argumentos de tipo* en un nombre de espacio de *nombres o-tipo* (§ 3,8). La instrucción
>
> ```
> F(G<A,B>(7));
> ```
>
> , según esta regla, se interpretará como una llamada a `F` con un argumento, que es una llamada a un método genérico `G` con dos argumentos de tipo y un argumento normal. Las instrucciones
>
> ```
> F(G < A, B > 7);
> F(G < A, B >> 7);
> ```
>
> cada uno de ellos se interpretará como una llamada a `F` con dos argumentos. La instrucción

```
x = F < A > +y;
```

se interpretará como un operador menor que, mayor que y unario más, como si la instrucción se hubiera escrito `x = (F < A) > (+y)`, en lugar de como un *nombre simple* con una *lista de argumentos de tipo* seguido de un operador binario Plus. En la instrucción

```
x = y is C<T> + z;
```

los tokens `C<T>` se interpretan como un *espacio de nombres o un nombre de tipo* con una *lista de argumentos de tipo*.

Hay una serie de cambios que se introducen C# en 7 que hacen que estas reglas de desambiguación dejen de ser suficientes para controlar la complejidad del lenguaje.

Declaraciones de variable out

Ahora es posible declarar una variable en un argumento out:

```
M(out Type name);
```

Sin embargo, el tipo puede ser genérico:

```
M(out A<B> name);
```

Dado que la gramática de idioma para el argumento usa *Expression*, este contexto está sujeto a la regla de desambiguación. En este caso, el `>` de cierre va seguido de un *identificador*, que no es uno de los tokens que permite que se trate como una *lista de argumentos de tipo*. Por lo tanto, se propone **Agregar el *identificador* al conjunto de tokens que desencadena la desambiguación en una *lista de argumentos de tipo*.**

Tuplas y declaraciones de desconstrucción

Un literal de tupla se ejecuta exactamente en el mismo problema. Considere la expresión de tupla.

```
(A < B, C > D, E < F, G > H)
```

En las 6 C# reglas anteriores para analizar una lista de argumentos, esto se analizaría como una tupla con cuatro elementos, empezando por `A < B` como primer. Sin embargo, cuando esto aparece a la izquierda de una desconstrucción, queremos que la desambiguación se desencadene por el token del *identificador* como se describió anteriormente:

```
(A<B,C> D, E<F,G> H) = e;
```

Se trata de una declaración de desconstrucción que declara dos variables, la primera de las cuales es de tipo `A<B,C>` y con el nombre `D`. En otras palabras, el literal de tupla contiene dos expresiones, cada una de las cuales es una expresión de declaración.

Para simplificar la especificación y el compilador, propongo que este literal de tupla se analice como una tupla de dos elementos dondequiera que aparezca (ya sea o no aparece en el lado izquierdo de una asignación). Esto sería un resultado natural de la desambiguación descrita en la sección anterior.

Coincidencia de patrones

La coincidencia de patrones introduce un nuevo contexto en el que se produce la ambigüedad del tipo de

expresión. Anteriormente, el lado derecho de un operador `is` era un tipo. Ahora puede ser un tipo o una expresión, y si es un tipo, puede ir seguido de un identificador. Técnicamente, esto puede cambiar el significado del código existente:

```
var x = e is T < A > B;
```

Esto podría analizarse en reglas de C# 6 como

```
var x = ((e is T) < A) > B;
```

pero bajo bajo las reglas de C# 7 (con la desambiguación propuesta anteriormente) se analizaría como

```
var x = e is T<A> B;
```

que declara una variable `B` de tipo `T<A>`. Afortunadamente, los compiladores nativo y Roslyn tienen un error por el que dan un error de sintaxis en el código de C# 6. Por lo tanto, este cambio importante concreto no es un problema.

La coincidencia de patrones introduce tokens adicionales que deben impulsar la resolución de ambigüedades al seleccionar un tipo. Los siguientes ejemplos de código de C# 6 válido existente se interrumpirían sin reglas de desambiguación adicionales:

```
var x = e is A<B> && f;            // &&
var x = e is A<B> || f;            // ||
var x = e is A<B> & f;             // &
var x = e is A<B>[];               // [
```

Cambio propuesto en la regla de desambiguación

Propongo revisar la especificación para cambiar la lista de tokens de ambigüedad de

```
( ) ] } : ; , . ? == != | ^
```

to

```
( ) ] } : ; , . ? == != | ^ && || & [
```

Y, en ciertos contextos, tratamos el *identificador* como un token de ambigüedad. Estos contextos son los lugares en los que la secuencia de los tokens que se van a la ambigüedad está inmediatamente precedida por una de las palabras clave `is`, `case` o `out`, o cuando se analiza el primer elemento de un literal de tupla (en cuyo caso los tokens van precedidos de `(` o `:` y el identificador va seguido de un `,`) o de un elemento subsiguiente de

Regla de desambiguación modificada

La regla de desambiguación revisada sería similar a la siguiente

> Si se puede analizar una secuencia de tokens (en contexto) como un *nombre simple* (§ 7.6.3), *acceso a miembro* (§ 7.6.5) o *acceso a miembro de puntero* (§ 18.5.2) que termina con una lista de *argumentos de tipo* (§ 4.4.1), se examina el token inmediatamente posterior al cierre `>` token para ver si es
>
> - Uno de `( ) ] } : ; , . ? == != | ^ && || & [`; de

- Uno de los operadores relacionales `< > <= >= is as`; de
- Una palabra clave de consulta contextual que aparece dentro de una expresión de consulta; de
- En ciertos contextos, tratamos el *identificador* como un token de ambigüedad. Estos contextos son donde la secuencia de tokens que se va a desambigüedadr está inmediatamente precedida por una de las palabras clave `is`, `case` o `out`, o bien al analizar el primer elemento de un literal de tupla (en cuyo caso los tokens van precedidos de `(` o `:` y el identificador va seguido de un `,`) o un elemento subsiguiente de un literal de tupla

Si el token siguiente se encuentra entre esta lista o un identificador en un contexto de este tipo, la *lista de argumentos de tipo* se conserva como parte del *nombre simple*, acceso a *miembros* o puntero a *miembro de puntero* , y se descarta cualquier otro posible análisis de la secuencia de tokens. De lo contrario, la *lista de argumentos de tipo* no se considera parte del acceso *simple*, de acceso a *miembros* o de *miembro de puntero*, incluso si no hay otro posible análisis de la secuencia de tokens. Tenga en cuenta que estas reglas no se aplican al analizar una *lista de argumentos de tipo* en un nombre de espacio de *nombres o-tipo* (§ 3,8).

Cambios importantes debidos a esta propuesta

No se conoce ningún cambio importante debido a esta regla de desambiguación propuesta.

Ejemplos interesantes

Estos son algunos resultados interesantes de estas reglas de desambiguación:

La expresión `(A < B, C > D)` es una tupla con dos elementos, cada una de las cuales es una comparación.

La expresión `(A<B,C> D, E)` es una tupla con dos elementos, el primero de los cuales es una expresión de declaración.

La `M(A < B, C > D, E)` de invocación tiene tres argumentos.

La `M(out A<B,C> D, E)` de invocación tiene dos argumentos, el primero de los cuales es una declaración de `out`.

La expresión `e is A<B> C` utiliza una expresión de declaración.

La etiqueta Case `case A<B> C:` usa una expresión de declaración.

Algunos ejemplos de coincidencia de patrones

Es como

Podemos reemplazar la expresión

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

Con el aspecto más conciso y directo

```
if (expr is Type v) {
    // code using v
}
```

Prueba de valores NULL

Podemos reemplazar la expresión

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

Con el aspecto más conciso y directo

```
if (x?.y?.z is Type value) {
    // code using value
}
```

Simplificación aritmética

Supongamos que definimos un conjunto de tipos recursivos para representar expresiones (según una propuesta independiente):

```
abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;
```

Ahora se puede definir una función para calcular el derivado (no reducido) de una expresión:

```
Expr Deriv(Expr e)
{
  switch (e) {
    case X(): return Const(1);
    case Const(*): return Const(0);
    case Add(var Left, var Right):
      return Add(Deriv(Left), Deriv(Right));
    case Mult(var Left, var Right):
      return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
    case Neg(var Value):
      return Neg(Deriv(Value));
  }
}
```

Un Simplificador de expresiones muestra patrones posicionales:

```
Expr Simplify(Expr e)
{
  switch (e) {
    case Mult(Const(0), *): return Const(0);
    case Mult(*, Const(0)): return Const(0);
    case Mult(Const(1), var x): return Simplify(x);
    case Mult(var x, Const(1)): return Simplify(x);
    case Mult(Const(var l), Const(var r)): return Const(l*r);
    case Add(Const(0), var x): return Simplify(x);
    case Add(var x, Const(0)): return Simplify(x);
    case Add(Const(var l), Const(var r)): return Const(l+r);
    case Neg(Const(var k)): return Const(-k);
    default: return e;
  }
}
```

Tutoriales de C#

18/03/2020 • 4 minutes to read • Edit Online

Los tutoriales proporcionan instrucciones paso a paso para escenarios comunes, lo que hace que un buen lugar para empezar a conocer el producto o una característica en particular.

Esta sección contiene vínculos a los tutoriales de programación C#.

En esta sección

- Acceso a la Web mediante async y await
  Muestra cómo crear una solución asincrónica mediante el uso de async y await.
- Creación de un componente de Windows Runtime en C# o Visual Basic, e invocación desde JavaScript
  Muestra cómo crear un tipo de Windows Runtime, empaquetarlo en un componente de Windows Runtime y, a continuación, llamar al componente desde una aplicación de la Tienda Windows 8.x creada para Windows mediante JavaScript.
- Programación de Office (C# y Visual Basic)
  Muestra cómo crear un libro de Excel y un documento de Word mediante C# y Visual Basic.
- Creación y uso de objetos dinámicos (C# y Visual Basic)
  Muestra cómo crear un objeto personalizado que expone dinámicamente el contenido de un archivo de texto y cómo crear un proyecto que utiliza la biblioteca `IronPython`.
- Creación de un control compuesto con Visual C#
  Muestra cómo crear un control compuesto simple y ampliar su funcionalidad mediante la herencia.
- Creación de un control de Windows Forms que aproveche las características en tiempo de diseño de Visual Studio
  Muestra cómo crear un diseñador personalizado para un control personalizado.
- Herencia de un control de Windows Forms con Visual C#
  Muestra cómo crear un control de botón heredado simple. Este botón hereda la funcionalidad del botón estándar de formularios Windows Forms y expone a un miembro personalizado.
- Depuración de controles personalizados de Windows Forms en tiempo de diseño
  Describe cómo depurar el comportamiento en tiempo de diseño del control personalizado.
- Tutorial: Realización de tareas comunes con acciones de diseño
  Muestra algunas de las tareas realizadas normalmente como agregar o quitar una pestaña en un `TabControl`, acoplar un control a su elemento primario y cambiar la orientación de un control `SplitContainer`.
- Escritura de consultas en C# (LINQ)
  Muestra las características del lenguaje C# que se usan para escribir expresiones de consulta de LINK.
- Manipulación de datos (C#) (LINQ to SQL)
  Describe un escenario LINQ to SQL para agregar, modificar y eliminar datos en una base de datos.
- Modelo de objetos simple y consultas (C#) (LINQ to SQL)
  Muestra cómo crear una clase de entidad y una consulta simple para filtrar la clase de entidad.
- Uso exclusivo de procedimientos almacenados (C#) (LINQ to SQL)

Muestra cómo utilizar LINQ to SQL para acceder a datos mediante la ejecución solamente de los procedimientos almacenados.

- Consultas entre relaciones (C#) (LINQ to SQL)
  Muestra el uso de asociaciones LINQ to SQL para representar relaciones de clave externa en una base de datos.

- Escritura de un visualizador en C#
  Muestra cómo escribir un visualizador sencillo con C#.

Secciones relacionadas

- Ejemplos y tutoriales de implementación
  Proporciona ejemplos detallados de escenarios comunes de implementación.

Vea también

- Guía de programación de C#
- Ejemplos de Visual Studio